CodeWarrior®

Pascal Library Reference

Because of last-minute changes to CodeWarrior, some of the information in this manual may be inaccurate. Please read the Release Notes on the CodeWarrior CD for the latest up-to-date information.

Revised: 98/08/17 map

Metrowerks CodeWarrior copyright ©1993–1998 by Metrowerks Inc. and its licensors. All rights reserved.

Documentation stored on the compact disk(s) may be printed by licensee for personal use. Except for the foregoing, no part of this documentation may be reproduced or transmitted in any form by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from Metrowerks Inc.

Metrowerks, the Metrowerks logo, CodeWarrior, and Software at Work are registered trademarks of Metrowerks Inc. PowerPlant and PowerPlant Constructor are trademarks of Metrowerks Inc.

All other trademarks and registered trademarks are the property of their respective owners.

ALL SOFTWARE AND DOCUMENTATION ON THE COMPACT DISK(S) ARE SUBJECT TO THE LICENSE AGREEMENT IN THE CD BOOKLET.

How to Contact Metrowerks:

U.S.A. and international	Metrowerks Corporation 9801 Metric, Suite 100 Austin, TX 78758 U.S.A.
Canada	Metrowerks Inc. 1500 du College, Suite 300 Ville St-Laurent, QC Canada H4L 5G6
Ordering	Voice: (800) 377–5416 Fax: (512) 873–4901
World Wide Web	`http://www.metrowerks.com`
Registration information	`register@metrowerks.com`
Technical support	`support@metrowerks.com`
Sales, marketing, & licensing	`sales@metrowerks.com`
CompuServe	goto `Metrowerks`

Table of Contents

1 Introduction **9**

Pascal Library Reference Overview 9
Read the Release Notes 9
What's New 10
What's in This Reference. 10
Routines Available in CodeWarrior Pascal 11
Predefined Routines 11
Mac OS Routines 13
Win32/x86 Routines 14
MSL C Routines 14
MPW Pascal Library Routines 15
Compatibility With Other Pascals 15
ANS Pascal 15
MPW Pascal 16
Turbo Pascal 16
Starting Points 16

2 Mathematics **19**

Mathematical Routines Overview 19
Limits and Accuracy 20
Mathematical Routines 20
Abs 20
Cos 21
Exp 22
Frac 23
Int 24
Ln 24
Pi 25
Random. 25
Randomize 25
RandSeed 26
Round 26
Sin 28
Sqr 29

SqRt 30
Trunc 31

3 MSL C in Pascal 33
MSL Routines in Pascal Overview 33
Choosing Units for MSL C 33
Configuring Your Project for MSL C 35
Input/output for MSL C 35
Strings for MSL C 36
Procedural Types and MSL C 36

4 Bit Manipulation 37
Bit Manipulation Routines Overview 37
Data Types 38
Bit Numbering 39
Mac OS and Bit Numbering 39
Bit Manipulation Routines 40
BAND 40
BClr 41
BNOT 43
BOR 43
BRotL 43
BRotR 45
BSet 46
BSL 46
BSR 47
BTst 47
BXOR 49
Hi 49
HiWrd 49
Lo 51
LoWrd 51
Swap 51

5 Byte & Array Manipulation 53
Byte and Array Manipulation Routines Overview 53

Byte and Array Manipulation Routines 54
FillChar 54
High 56
Move 57
MoveLeft 58
MoveRight 60
Pack 60
ScanEq 61
ScanNE 62
Unpack 62

6 Exit and Halt Routines 65
Exit and Halt Routines Overview 65
Exit and Halt Routines 65
Exit. 65
Halt 67
RunError 68

7 Input and Output 69
Input/Output Routines Overview 69
Routines for All Files 69
Routines for Structured and Untyped Files. 70
Routines for Structured and Text Files. 71
Routines for Raw Data File Access 71
Routines for Text File Access 71
Choosing Which I/O Routines to Use 73
Input and Output Routines 76
Append 76
Assign 76
BlockRead 77
BlockWrite 81
ByteRead 83
ByteWrite 85
Close 86
EOF 88
EOLn 90

Erase 91
FileMode 92
FilePos 93
FileSize 93
Flush 94
Get 94
InOutRes 96
IOResult 97
Open 99
Page 100
PLCrunch 100
PLFilePos 100
PLFlush 101
PLPurge 101
PLRename 102
PLSetVBuf 102
Put 103
Read 104
ReadLn 111
Rename 112
Reset 112
Rewrite 115
Seek 117
SeekEOF 118
SeekEOLn 119
SetTextBuf 119
Truncate 120
Write 120
WriteLn 127

8 Console Input & Output on Mac OS 129

SIOUX Overview 129
Creating a Project With SIOUX 131
Customizing SIOUX 132
SIOUXSettings 132

Changing the Font and Tabs 135
Changing the Size and Location 136
Changing What Happens on Quit 137
Showing the Status Line 138
Using SIOUX Windows in Your Own Application 138
SIOUXHandleOneEvent 139
SIOUXSetTitle 140

9 Memory & Object Manipulation 143
Memory Management Routines Overview 143
Memory Management Routines 144
Dispose 144
FreeMem 145
GetMem 146
HeapResult 146
Mark 148
MaxAvail 148
MemAvail 149
Member 149
New 151
PLHeapInit 155
PLSetHeapCheck 156
PLSetHeapType 157
PLSetMErrProc 157
PLSetNonCont 158
Release 158
SizeOf 159

10 Ordinal Operations 161
Ordinal Routines Overview 161
Ordinal Operation Routines 161
Addr 161
Chr 163
Dec 164
Inc 166
Odd 166

Ord 167
Ord4 169
Pointer 171
Pred 171
Succ 172
UpCase 173

11 String Manipulation 175
String Manipulation Routines Overview 175
String Manipulation Routines 176
Concat 176
Copy 177
Delete 179
Insert 180
Length 181
PLStringOfGet 182
PLStringOfSet 183
Pos 184
ReadString 184
Str 185
StringOf. 186
StrOf 188
Val 191

Index 193

1 Introduction

This reference describes the library routines available for programming in CodeWarrior Pascal.

Pascal Library Reference Overview

This reference describes the pre-defined routines that are available through the CodeWarrior Pascal runtime library.

For more information about using predefined and library routines in CodeWarrior Pascal, refer to these topics:

- Read the Release Notes—getting last-minute information about CodeWarrior Pascal's library routines.
- What's New—what's been changed in and added to this reference.
- What's in This Reference—descriptions of the chapters in this manual
- Routines Available in CodeWarrior Pascal—an overview of the kinds of routines available
- Compatibility With Other Pascals—choosing an I/O model to use predefined routines from other Pascal implementations
- Starting Points—where to look for other information on CodeWarrior Pascal library routines.

Read the Release Notes

Before using CodeWarrior Pascal, read the documentation in the `Release Notes` folder. The notes in this folder contain important information about new features, bug fixes, and any late-breaking changes.

What's New

New and updated topics in this reference are:

- "MPW Pascal Library Routines" on page 15—new information on using "PL" routines
- "Input and Output" on page 69—corrected prototypes for these routines: `PLCrunch`, `PLFilePos`, `PLFlush`, `PLPurge`, `PLRename`, `PLSetVBuf`
- "IOResult" on page 97—new information on deciphering an error value returned by `IOResult`.
- "Memory & Object Manipulation" on page 143—corrected prototypes for these routines: `PLHeapInit`, `PLSetHeapCheck`, `PLSetHeapType`, `PLSetMErrProc`, `PLSetNonCont`

What's in This Reference

Table 1.1 lists the chapters in this reference and the topics they cover.

Table 1.1 Descriptions of the chapters in this reference

Refer to this chapter…	To find out about
"Pascal Library Reference Overview" on page 9	this reference
"Mathematical Routines Overview" on page 19	arithmetic and floating point routines
"MSL Routines in Pascal Overview" on page 33	using Metrowerks Standard Library for C (MSL C) routines in Pascal
"Bit Manipulation Routines Overview" on page 37	logical and bit handling routines
"Byte and Array Manipulation Routines Overview" on page 53	routines for filling, moving, and searching memory

Refer to this chapter…	To find out about
"Exit and Halt Routines Overview" on page 65	routines for stopping execution in a statement block
"Input/Output Routines Overview" on page 69	routines for reading, writing, and manipulating files and console data
"SIOUX Overview" on page 129	routines for console input/output
"Memory Management Routines Overview" on page 143	routines for allocating, deallocating, and manipulating heap memory and objects
"Ordinal Routines Overview" on page 161	routines for operating on scalar and pointer values
"String Manipulation Routines Overview" on page 175	routines for handling character strings

Routines Available in CodeWarrior Pascal

Many routines and libraries are included for use in CodeWarrior Pascal:

- Predefined Routines—routines that treated as part of the CodeWarrior Pascal language.
- Win32/x86 Routines—routines that access Microsoft's Win32 API for Windows 95, Windows NT, and later versions of these operating systems.
- Mac OS Routines—routines that access the Mac OS.
- MSL C Routines—routines in the MSL C accessible from CodeWarrior Pascal
- MPW Pascal Library Routines—routines in the MPW Pascal Library

Predefined Routines

This reference documents the predefined routines that are available while programming with CodeWarrior Pascal and some routines that are not predefined but are available in external libraries that

may be added to your project. The predefined functions and procedures are considered part of the CodeWarrior Pascal language. They are in the CodeWarrior Pascal runtime library.

Declaring Predefined Routines

You are not required to (and cannot) explicitly specify a unit that declares the predefined routines, although you must add the appropriate runtime library to your project.

NOTE: Runtime libraries are automatically in your project when you create a new one using the **New Project** command from the CodeWarrior IDE's File menu.

These functions and procedures are predeclared. Predefined objects act as if they were declared in a block surrounding the program, so no conflict arises from a declaration that redefines the same identifier within the program. Therefore new identifiers that have the same name as predefined identifiers take precedence over the predefined identifiers.

NOTE: Predefined procedures and functions cannot be used as actual parameters for procedures and functions.

When Predefined and New Identifiers Conflict

If the name of a predefined object is reused in your project, you can refer to the original predefined object using qualifier identification. Refer to the original predefined item by qualifying it with "`System.`".

For example, the routine `Round`, predefined in CodeWarrior Pascal, is also defined in the Mac OS Universal Interface file `fp.p`. If your project uses the `fp.p` Universal Interface file and you want to refer to the predefined `Round` routine, enter

```
System.Round(x)
```

This statement calls the predefined Round routine, not the routine defined in `fp.p`.

To refer to the Round routine declared in `fp.p`, use

```
Round(x)
```

or

```
fp.Round(x)
```

For more information on qualifier identification refer to the CodeWarrior *Pascal Language Reference*.

Mac OS Routines

Some versions of CodeWarrior provide interfaces and libraries to access the Mac OS. These interfaces are the Universal Programming Interfaces (UPI). They are universal because the same interface files can be used to generate source code for Mac OS computers with PowerPC or Motorola 68K processors.

For documentation on Mac OS, refer to this documentation:

- *Inside Macintosh* volumes. Apple Computer Inc., published by Addison-Wesley.
- Apple's software developer Web page:

  `http://www.apple.com/developer`

  This web page contains on-line versions of *Inside Macintosh*, and other software development documentation and examples.

NOTE: Libraries and access paths for using the Mac OS are automatically added when you create a new project for Mac OS using the **New Project** command. Runtime libraries for predefined Pascal routines are also added automatically.

Win32/x86 Routines

Some versions of CodeWarrior provide interfaces and libraries to access the Windows 95, Windows NT, and later Windows operating systems running on Intel x86-compatible processors. These interfaces form the Microsoft Win32 API.

A single interface file, `Windows.p`, provides access to this API.

For documentation on the Win32 API, refer to this documentation:

- Microsoft's software development Web pages:

  `http://www.microsoft.com/win32dev`

  `http://www.microsoft.com/msdn`
- Other 3rd party books that cover programming for Windows 95 and Windows NT, or later versions of these operating systems.

NOTE: Libraries and access paths for using the Win32 API are automatically added to a project when you create a new one for Win32/x86 using the **New Project** command. Runtime libraries for predefined Pascal routines are also added automatically.

MSL C Routines

You may access MSL C (Metrowerks Standard Library for C) routines through special Pascal Interfaces.

For more information on using MSL C in CodeWarrior Pascal, see "MSL Routines in Pascal Overview" on page 33.

NOTE: Libraries and access paths for using MSL C in Pascal are automatically added to a project when you create a new one using the **New Project** command. Runtime libraries for predefined Pascal routines are also added automatically.

MPW Pascal Library Routines

Some routines that begin with "PL" are part of the MPW Pascal Library. These routines are only available when compiling for Mac OS. To use them add `PasLibIntf` to the `USES` clause of your program or unit.

These routines are:

`PLCrunch`
`PLFilePos`
`PLFlush`
`PLPurge`
`PLRename`
`PLSetVBuf`
`PLHeapInit`
`PLSetHeapCheck`
`PLSetHeapType`
`PLSetMErrProc`
`PLSetNonCont`

Compatibility With Other Pascals

CodeWarrior Pascal provides routines that are compatible with the predefined routines in other Pascal implementations. To compile source code that refers to predefined routines available in other Pascals, choose an item in the **IO Model** popup menu in the **Pascal Language** settings panel.

The items in this popup menu are:

- ANS Pascal
- MPW Pascal
- Turbo Pascal

For related information on choosing an I/O model, see "Choosing Which I/O Routines to Use" on page 73.

ANS Pascal

Makes ANS Pascal predefined routines available. This set of I/O routines is also known as Standard Pascal routines or BSI Pascal

(British Standards Pascal) routines. Use this I/O model when compiling Pascal source code that was written to run on any computer platform. The routines in this model are found in most Pascal implementations.

MPW Pascal

Makes MPW Pascal predefined routines available. This set of I/O routines matches the set of I/O routines in Apple Computer's MPW Pascal (Macintosh Programmer's Workshop Pascal).

Use this I/O model when compiling Pascal source code that was written for MPW Pascal.

See also "MPW Pascal Library Routines" on page 15.

Turbo Pascal

This set of I/O routines matches the set of I/O and other routines provided by Borland's Turbo Pascal version 7.0 compiler.

Use this I/O model when compiling Pascal source code that was written for Turbo Pascal.

Starting Points

The following manuals and books contain information that you'll find useful when writing programs in CodeWarrior Pascal.

Pascal Language Reference

This documentation describes the CodeWarrior implementations of the Pascal programming language.

CodeWarrior IDE User Guide

This manual describes the CodeWarrior IDE, the development environment in which CodeWarrior Pascal runs.

CodeWarrior Porting Guide

Use this manual to port a Pascal program from another development environment to CodeWarrior. The *CodeWarrior Porting Guide* offers tips, hints, and references to other documentation.

2

Mathematics

The mathematical routines provide mathematical operations for floating point numbers (`REAL`, `LONGREAL`, `SINGLE`, or `DOUBLE` data types) and other numerical operations.

Mac OS For more mathematical operations, use the routines available through operating system interfaces, such as the `fp.p` Universal Interface for Mac OS.

Mathematical Routines Overview

The mathematical routines available through the CodeWarrior Pascal runtime library are:

`Abs`	Computes an absolute value.
`Cos`	Computes a cosine value.
`Exp`	Computes a natural exponent.
`Frac`	Returns the fractional part of a number.
`Int`	Removes the fractional part of a number.
`Ln`	Computes the natural logarithm.
`Pi`	Returns the value of pi.
`Random`	Returns a pseudo-random number.
`Randomize`	Initializes the pseudo-random number generator.
`RandSeed`	Holds the random number seed global variable.
`Round`	Rounds off a number.
`Sin`	Computes a sine value.

Sqr	Computes the square of a value.
SqRt	Computes the square root.
Trunc	Removes the fractional part of a number

To find out about the numerical properties of the mathematical routines in the CodeWarrior Pascal runtime library, see "Limits and Accuracy" on page 20.

Limits and Accuracy

For information about the limits and accuracy of these routines, consult the MSL C and MSL C++ references. Pascal math library routines are based on MSL.

Mathematical Routines

Abs

Description Computes the absolute value.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
FUNCTION Abs(x : NumericalType) : NumericalType;
```

Remarks The `Abs` function returns the absolute value of a numeric value. Its parameter *x* is a numeric value. *NumericalType* represents a numerical data type. Other scalar types, such as `BOOLEAN` and enumerated types, are not valid.

The expression

```
Abs(-MaxLONGINT - 1)
```

produces an overflow because the value

```
-MaxLONGINT - 1
```

cannot be expressed as an absolute value using the `LONGINT` data type.

Returns `Abs(`*x*`)` returns the absolute value of *x*, with the same type.

See also Odd

Listing 2.1 Abs example

```
PROGRAM Abs_Example(input, output);

VAR
  i : INTEGER;
  x : REAL;

BEGIN
  i := -10;
  x := 3.14159;
  WriteLn(i:10, Abs(i):10, x:10:5, Abs(x):10:5);
END.

(* Output:

       -10        10   3.14159   3.14159
*)
```

Cos

Description Computes the cosine.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
FUNCTION Cos(x : RealType) : RealType;
```

Remarks The `Cos` function returns a value that is the cosine of a numeric value. Its parameter *x* is a numeric value. *RealType* represents a floating point data type.

The *x* value is assumed to represent an angle in radians.

Returns Cos returns the cosine of its argument.

See also Sin

Listing 2.2 Cos example

```
PROGRAM Cos_example(input, output);

VAR
  x : REAL;

BEGIN
  x := 0.5;

  WriteLn('cos(', x, ') = ', Cos(x));
END. { Cos_example }

(* Output:

cos(  0.500000) =   0.877583

*)
```

Exp

Description Computes a natural exponential value.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
FUNCTION Exp(x : RealType) : RealType;
```

Remarks The Exp function returns a value that is the natural exponential of a numeric value. Its parameter *x* is a numeric value. *RealType* represents a floating point data type. All possible values are valid.

Returns `Exp(x)` returns the value of e^x, where e is the base of the natural logarithm. If floating point overflow occurs, the result is `+inf`.

See also `Ln`

Listing 2.3 Exp example

```
PROGRAM Exp_Example(input, output);
VAR
  e : LONGREAL;

BEGIN
  e := Exp(1.0);
  WriteLn(Ln(e):10:10);
END.

(* Output:

1.0000000000

*)
```

Frac

Description Returns the fractional part of a floating point value.

Compatibility Turbo Pascal

Prototype

```
FUNCTION Frac(x : RealType) : RealType;
```

Remarks The `Frac` function returns the fractional part of a real numeric value. *RealType* represents a floating point data type. All possible values are valid.

Returns `Frac(x)` returns the value

```
x - Trunc(x)
```

See also `Int`, `Round`, Listing 2.4

Int

Description Returns the integer part of a floating point value.

Compatibility Turbo Pascal

Prototype

```
FUNCTION Int(x : RealType) : RealType;
```

Remarks The `Int` function converts a floating point value to a `LONGINT` value. Its parameter *x* is an expression with a value of type represented by *RealType*, `LONGREAL` or `REAL`, depending on the expression type the function is in.

Returns `Int(`*x*`)` returns a result that is the value of *x* rounded to the largest whole number between zero and *x* (inclusive).

See also `Round`, Listing 2.4

Ln

Description Computes the natural logarithm.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
FUNCTION Ln(x : RealType) : RealType;
```

Remarks The `Ln` function returns a value that is the natural logarithm of a numeric value. Its parameter *x* is a numeric value. *RealType* represents a floating point data type.

Returns `Ln(`*x*`)` returns the natural logarithm ($\log_e$) of *x* if *x* is non-negative. If *x* is negative, a diagnostic `NAN` is produced and the invalid operation signal is set.

See also `Exp`, Listing 2.3.

Pi

Description Returns the value of pi.

Compatibility Turbo Pascal

Prototype

```
FUNCTION Pi : RealType;
```

Remarks *RealType* represents a floating point data type.

Returns Returns 3.1415926535897932385

Random

Description Rounds off a real value to a `LONGINT` value.

Compatibility Turbo Pascal

Prototype

```
FUNCTION Random [(range : IntegerType) ] : REAL;

FUNCTION Random(range : IntegerType) : LONGINT;
```

Remarks Before calling `Random`, make sure to call `Randomize` to ensure that a new sequence of pseudo-random numbers are generated.

Returns The `Random` function returns a pseudo-random value greater or equal to 0 and less than range. If no parameter is passed to `Random`, the default value of *range* is 1.

See also `Randomize`

Randomize

Description Initializes the random number generator.

Compatibility Turbo Pascal

Prototype

```
PROCEDURE Randomize;
```

Remarks The `Randomize` routine initializes the pseudo-random number generator function, ensuring that a fresh sequence of random numbers will be generated. To generate a random number after calling `Randomize`, call Random.

The `Randomize` routine initializes the pseudo-random number generator by assigning a random value to `RandSeed`. This random value is derived from the current date and time at runtime.

See also `Random`

RandSeed

Description Contains the value to generate pseudo-random numbers from.

Compatibility Turbo Pascal

Prototype

```
RandSeed : LONGINT;
```

Remarks The `RandSeed` global variable contains the value with which pseudo-random numbers are generated from. Assigning a random value, often a value based on the current date or time, to RandSeed ensures that a fresh series of pseudo-random numbers will be generated when calling `Random`. To repeat a series of pseudo-random numbers, assign the same value to RandSeed before making calls to `Random`.

See also `Random`, `Randomize`

Round

Description Rounds off a real value to a `LONGINT` value.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
FUNCTION Round(x : RealType) : LONGINT;
```

Remarks The Round function converts a real value to a LONGINT value. Its parameter *x* is an expression with a value of type represented by *RealType*. *RealType* can be REAL or LONGREAL. If *x* is exactly halfway between two whole numbers, the result is the whole number with the greatest absolute magnitude.

Returns Round rounds off a real value, returning it as a LONGINT value.

See also Frac, Trunc

Listing 2.4 Round, Frac, and Trunc example

```
PROGRAM Round_Example(input, output);

BEGIN
  WriteLn( Round(1234.56789) );
  WriteLn( Trunc(1234.56789) );
  WriteLn( Round(-1234.4321) );
  WriteLn( Trunc(-1234.4321) );
  WriteLn( Int(-1234.4321) );
  WriteLn( Frac(1234.4321) );
END. { Round_Example }

(* Output:

1235
1234
-1234
-1234
-1234
0.4321

*)
```

Sin

Description Computes the sine.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
FUNCTION Sin(x : RealType) : RealType;
```

Remarks The `Sin` function returns a value that is the sine of a numeric value. Its parameter *x* is a numeric value. *RealType* represents a floating point data type. This value is assumed to represent an angle in radians. If *x* is infinite, a diagnostic `NAN` is produced and the invalid operation signal is set.

Returns `Sin` returns the sine of its parameter as a `LONGREAL` or `REAL` value, depending on the expression type the function is in.

See also `Cos`

Listing 2.5 Sin example

```
PROGRAM Sin_example(input, output);

VAR
  x : REAL;

BEGIN
  x := 0.5;

  WriteLn('sin(', x:10:10, ') = ', Sin(x):10:10);
END. { Sin_example }

(* Output:

sin(0.5000000000) = 0.2147483647

*)
```

Sqr

Description Computes the square of a value.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
FUNCTION Sqr(x : NumericalType) : NumericalType;
```

Remarks The `Sqr` function returns the square of a numeric value. Its parameter *x* is a numeric value.

If *x* is a floating point type, the result is a floating point value; if `NumericalType` is of type `LONGINT`, the result is `LONGINT`; and if `NumericalType` is of type `INTEGER`, the result may be either `INTEGER` or `LONGINT`.

If *x* is of floating point type and floating point overflow occurs, the result is `+inf`.

Returns `Sqr(`*x*`)` returns the square of *x*.

See also `SqRt`

Listing 2.6 Sqr example

```
PROGRAM Sqr_Example(input, output);

BEGIN
  { Real square }
  WriteLn('Square of 100: ', Sqr(100.0));

  { Integer square }
  WriteLn('Square of 100: ', Sqr(100));

  { Long integer square }
  WriteLn('Square of 100: ', Sqr(Ord4(100)));
END. { Sqr_Example }

(* Output:
```

```
Square of 100:  1.0e+0004
Square of 100: 10000
Square of 100: 10000

*)
```

SqRt

Description Computes the square root.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
FUNCTION SqRt(x : NumericalType) : RealType;
```

Remarks The `SqRt` function returns a value that is the square root of a numeric value. Its parameter *x* is a numeric value. *NumericalType* represents any numerical data type. *RealType* represents a floating point data type.

Returns `SqRt(`*x*`)` returns the positive square root of *x* if *x* is non-negative. If *x* is negative, a diagnostic `NAN` is produced and the invalid operation signal is set.

See also `Sqr`

Listing 2.7 SqRt example

```
PROGRAM SqRt_Example(input, output);

BEGIN
  { Real square root }
  WriteLn('Square root of 100: ', SqRt(100.0));

  { Integer square root }
  WriteLn('Square root of 100: ', SqRt(100));

  { Long integer square root }
  WriteLn('Square root of 100: ', SqRt(Ord4(100)));
```

```
END. { SqRt_Example }

(* Output:

Square root of 100:  1.0e+0001
Square root of 100:  1.0e+0001
Square root of 100:  1.0e+0001

*)
```

Trunc

Description Converts a real value to an integer value.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
FUNCTION Trunc(x : RealType) : LONGINT;
```

Remarks The `Trunc` function converts a floating point value to a `LONGINT` value. Its parameter *x* is an expression with a value of type represented by *RealType*, `LONGREAL` or `REAL`, depending on the expression type the function is in.

Returns `Trunc(`*x*`)` returns a `LONGINT` result that is the value of *x* rounded to the largest whole number between zero and *x* (inclusive).

See also `Int`, `Round`, Listing 2.4.

3

MSL C in Pascal

CodeWarrior Pascal has interface files that allow you to access most of the Metrowerks Standard Library for C (MSL C), which provides ANSI C Standard Library and many common UNIX™ functions from within CodeWarrior Pascal.

For descriptions of specific MSL C library routines, consult the MSL C Library Reference.

MSL Routines in Pascal Overview

This chapter shows you how to call Metrowerks Standard Library for C (MSL C) routines in CodeWarrior Pascal. The topics in this section are:

- Choosing Units for MSL C
- Configuring Your Project for MSL C
- Input/output for MSL C
- Strings for MSL C
- Procedural Types and MSL C

Choosing Units for MSL C

Edit the `USES` clause of your program to include the MSL C Library units you require. The MSL C unit names and their MSL C header file equivalents are listed in Table 3.1.

Table 3.1 Pascal interfaces and MSL C header files

To use the routines in this MSL C header file…	Add this unit in your USES clause
ansi_parms.h	ansi_parms
assert.h	assert
ctype.h	ctype
div_t.h	div_t
errno.h	errno
fcntl.h	fcntl
fenv.h	not available
float.h	float
limits.h	limits
locale.h	locale
math.h	math
null.h	null
setjmp.h	setjmp
signal.h	signal
size_t.h	size_t
stat.h	stat
stdarg.h	stdarg
stddef.h	stddef
stdio.h	stdio
stdlib.h	stdlib
string.h	string
time.h	time
unistd.h	unistd

To use the routines in this MSL C header file…	Add this unit in your USES clause
`unix.h`	`unix`
`utime.h`	`utime`
`va_list.h`	`va_list`
`wchar_t.h`	`wchar_t`

Configuring Your Project for MSL C

When using MSL C routines in CodeWarrior Pascal, make sure to use the proper prefix file. This prefix file should be included at the beginning of every Pascal source code file that refers to MSL C routines.

Windows When compiling for Win32/x86, use the `ANSI.TargetIntel.prefix` file.

Mac OS When compiling for Mac OS (PowerPC or 68K), use the `ANSI.TargetMac.prefix` file.

There are two ways to include the MSL C prefix file in your source code. The first method is to insert a `{$I filename}` directive at the beginning of each Pascal source code file that refers to MSL C routines, where *filename* is the name of the MSL C prefix file. The second method is to enter the name of the MSL C prefix file in the **Prefix File** field of the **Pascal Language** settings panel in your project.

Input/output for MSL C

The MSL C I/O functions, provided by `stdio.h`, and the Pascal I/O functions are not compatible. You cannot pass an MSL C `FILE` variable to a Pascal standard I/O routine and you cannot pass a Pascal file variable to a MSL C I/O routine. However, you can use C and Pascal I/O routines on different files at the same time.

The Standard C Library `printf()`, and `scanf()` functions (and their variants) are not supported.

Strings for MSL C

You cannot pass Pascal character strings (items of type STRING) to MSL C routines. However, you can pass C-friendly strings in a PACKED ARRAY OF CHAR.

Procedural Types and MSL C

MSL C functions which require call-back routines as a function argument are currently supported only for the Mac OS PowerPC because Pascal and C calling conventions on the PowerPC are the same.

4

Bit Manipulation

Bit manipulation functions and procedures are perform logical operations and tests on individual bits within items up to 32 bits large.

Bit Manipulation Routines Overview

CodeWarrior Pascal has these routines to test and set individual bits:

BAND	Performs a bit AND operation.
BClr	Performs bit clear operation.
BNOT	Performs bit NOT operation.
BOR	Performs bit OR operation.
BRotL	Rotates bits left.
BRotR	Rotate bits right
BSet	Sets a bit.
BSL	Shifts bits left.
BSR	Shifts bits right.
BTst	Checks a bit's value.
BXOR	Performs a bit exclusive-OR operation.
Hi	Gets upper 16 bits of a 32 bit value.
HiWrd	Gets upper 16 bits of a 32 bit value.
Lo	Gets lower 16 bits of a 32 bit value.
LoWrd	Gets lower 16 bits of a 32 bit value.
Swap	Exchanges the high and low order bytes in a value

To find out about how these routines treat their arguments, see "Data Types" on page 38.

To find out about how the bit numbering convention used by these routines, see "Bit Numbering" on page 39.

Data Types

If the type of any argument is specified as a scalar, the argument can be a whole-number value of any size, from 1 to 32 bits (one bit to a long integer). If the scalar argument is less than 32 bits, code is generated to extend the argument to 32 bits but without sign extension (zeros are added on the left to make up a 32-bit value).

Table 4.1 summarizes the bit manipulation functions and procedures.

Table 4.1 Bit manipulation routines

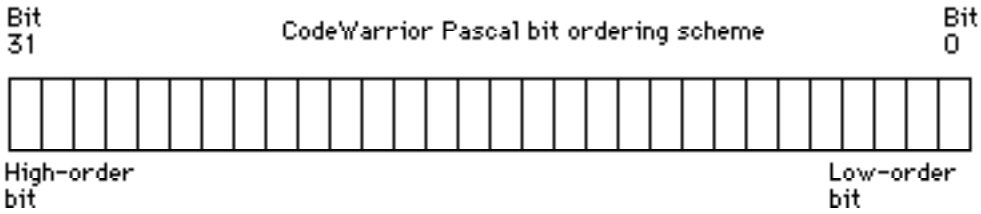
Name	First argument	Second argument	Result	Kind
BAND	scalar	scalar	LONGINT	function
BOR	scalar	scalar	LONGINT	function
BXOR	scalar	scalar	LONGINT	function
BNOT	scalar	none	LONGINT	function
BSL	scalar	INTEGER	LONGINT	function
BSR	scalar	INTEGER	LONGINT	function
BRotL	scalar	INTEGER	LONGINT	function
BRotR	scalar	INTEGER	LONGINT	function
BTst	scalar	INTEGER	BOOLEAN	function
BClr	LONGINT (VAR)	INTEGER	none	procedure
BSet	LONGINT (VAR)	INTEGER	none	procedure

Name	First argument	Second argument	Result	Kind
`HiWrd`	scalar	none	`INTEGER`	function
`LoWrd`	scalar	none	`INTEGER`	function

Bit Numbering

CodeWarrior Pascal bit manipulation routines follow the same numbering convention for all processors, including Intel x86-compatible, Motorola 680x0, and PowerPC processors: bit 0 is the low-order bit and bit 31 is the high-order bit.

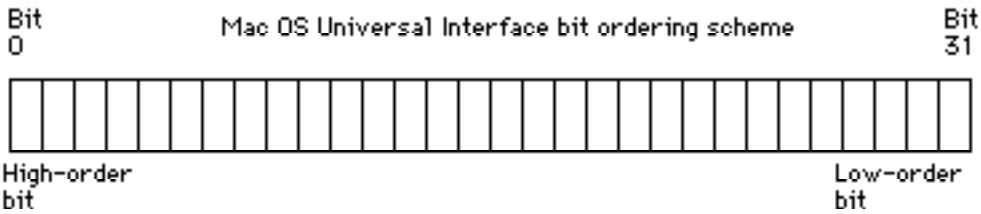
Figure 4.1 Bit order for CodeWarrior Pascal bit manipulation routines



Mac OS and Bit Numbering

Bit numbering for the bit manipulation routines declared in the `ToolUtils.p` Mac OS Universal Interface (`BitAnd`, `BitClr`, `BitNot`, `BitOr`, `BitSet`, `BitShift`, `BitTst`, `BitXor`, `LoWord`, and `HiWord`) begins at the high-order bit and increases to the low-order bit (Figure 4.2).

Figure 4.2 Bit order for Mac OS Universal Interface routines



The CodeWarrior Pascal bit manipulation routines are generally similar in name and function to the corresponding set of routines described in *Inside Macintosh: Operating System Utilities* and found in the Mac OS Universal Interfaces. However, the CodeWarrior Pascal routines are more efficient because they are implemented by the compiler as 680x0 or PowerPC 60x instructions, while the *Inside Macintosh* routines are calls to the Mac OS.

Bit Manipulation Routines

BAND

Description Performs a bit-wise, logical AND operation.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
FUNCTION BAND(
  arg1,
  arg2 : ScalarType) : LONGINT;
```

Remarks BAnd returns the logical AND of its two arguments.

If *arg1* or *arg2* is less than 32 bits, code is generated to extend the arguments to 32 bits without sign extension.

See also BNOT, BOR, BXOR

Listing 4.1 BAND, BOr, BXOr, and BNot example

```
PROGRAM BAnd_Example(input, output);

BEGIN
  WriteLn('1 AND 1 = ', BAND(1, 1):10);
  WriteLn('1 OR 0 = ', BOR(1, 0):10);
  WriteLn('1 XOR 0 = ', BXOR(1, 0):10);
  WriteLn('2''s complement of 53 = ', BNOT(53) + 1:10);
END.
```

```
(* Output:

1 AND 1 =           1
1 OR 0 =           1
1 XOR 0 =           1
2's complement of 53 =        -53

*)
```

BClr

Description Clears a bit.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
PROCEDURE BClr(
  VAR arg   : LONGINT;
  bitNbr    : IntegerType);
```

Remarks `BClr` clears bit *bitNbr* in *arg*. The value of *bitNbr* is reduced modulo 32. Bit 0 is considered the lowest-order bit, bit 31 the highest-order bit.

The *arg* parameter must be of type `LONGINT`. *BitNbr* can be any integer value.

If *arg* is less than 32 bits, code is generated to extend the argument to 32 bits without sign extension.

See also `BSet`, `BTst`

Listing 4.2 BClr and BSet example

```
PROGRAM BClr_Example(input, output);

VAR
  val : LONGINT;
  bitcount : INTEGER;
```

```
BEGIN
  val := 0;

  { Set bits, one at a time }
  FOR bitcount := 0 to 7 DO
  BEGIN
    WriteLn(val:10);
    BSet(val, bitcount);
  END;

  { Clear bits, one at a time }
  FOR bitcount := 7 DOWNTO 0 DO
  BEGIN
    WriteLn(val:10);
    BClr(val, bitcount);
  END;
END. { BClr_Example }

(* Output:

         0
         1
         3
         7
        15
        31
        63
       127
       255
       127
        63
        31
        15
         7
         3
         1

*)
```

BNOT

Description Performs 1's-complement bit-wise operation.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
FUNCTION BNOT(arg : ScalarType) : LONGINT
```

Remarks If *arg* is less than 32 bits, code is generated to extend the argument to 32 bits without sign extension.

Returns `BNOT` returns the 1's-complement of its argument.

See also `BAND`, `BOR`, `BXOR`, Listing 4.1.

BOR

Description Performs a bit-wise, logical OR operation.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
FUNCTION BOR(
  arg1,
  arg2 : ScalarType) : LONGINT;
```

Remarks If *arg1* or *arg2* is less than 32 bits, code is generated to extend the arguments to 32 bits without sign extension.

Returns `BOR` returns the logical `OR` of its two arguments.

See also `BAND`, `BNOT`, `BXOR`, Listing 4.1.

BRotL

Description Performs a left-rotate operation.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
FUNCTION BRotL(
  arg    : ScalarType;
  count  : INTEGER) : LONGINT;
```

Remarks If *arg* is less than 32 bits, code is generated to extend the argument to 32 bits without sign extension.

Returns `BRotL` left-rotates the bits of *arg* by the number of bits specified in *count*, modulo 64. Bits shifted out of the high-order position go back into the low-order position.

See also BRotR, BSL, BSR

Listing 4.3 BRotL, BRotR, BSL, and BSR example

```
PROGRAM BRotL_ex(input, output);

VAR
  count : INTEGER;
  brotlVal,
  brotrVal,
  bslVal,
  bsrVal : LONGINT;

BEGIN
  { Initialize the values to be manipulated }
  brotlVal := 128;
  brotrVal := 128;
  bslVal := 128;
  bsrVal := 128;

  { Print a column header }
  WriteLn('BRotL':12, 'BRotR':12, 'BSL':12, 'BSR':12);

  { Shift and rotate away! }
  FOR count :=1 to 10 DO
  BEGIN
    WriteLn(brotlVal:12, brotrVal:12, bslVal:12, bsrVal:12);
    brotlVal := BRotL(brotlVal, 1);
    brotrVal := BRotR(brotrVal, 1);
```

```
    bslVal := BSL(bslVal, 1);
    bsrVal := BSR(bsrVal, 1);
  END;
END.

(* Output:

       BRotL       BRotR         BSL         BSR
         128         128         128         128
         256          64         256          64
         512          32         512          32
        1024          16        1024          16
        2048           8        2048           8
        4096           4        4096           4
        8192           2        8192           2
       16384           1       16384           1
       32768 -2147483648       32768           0
       65536  1073741824       65536           0

*)
```

BRotR

Description Performs a right-rotate operation.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
FUNCTION BRotR(
  arg   : ScalarType;
  count : INTEGER) : LONGINT;
```

Remarks If *arg* is less than 32 bits, code is generated to extend the argument to 32 bits without sign extension.

Returns `BRotR` right-rotates the bits of *arg* by the number of bits specified in *count*, modulo 64. Bits shifted out of the low-order position go back into the high-order position.

See also BRotL, BSL, BSR, Listing 4.3.

BSet

Description Sets a bit.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
PROCEDURE BSet(
  VAR arg : LONGINT;
  bitNbr  : IntegerType);
```

Remarks If *arg* is less than 32 bits, code is generated to extend the argument to 32 bits without sign extension. The value of the bit to set, *bitNbr*, is reduced modulo 32. Bit 0 is considered the lowest-order bit, bit 31 the highest-order bit.

Returns `BSet` sets bit *bitNbr* in *arg*.

See also BClr, BTst, Listing 4.2.

BSL

Description Performs a left bit shift operation.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
FUNCTION BSL(
  arg   : ScalarType;
  count : IntegerType) : LONGINT;
```

Remarks If *arg* is less than 32 bits, code is generated to extend the argument to 32 bits without sign extension.

Returns `BSL` left-shifts the bits of *arg* by the number of bits specified in *count*, modulo 64. Zeros are shifted into the low-order bit.

See also BRotL, BRotR, BSR, Listing 4.3.

BSR

Description Performs a right bit shift operation.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
FUNCTION BSR(
  arg   : ScalarType;
  count : IntegerType) : LONGINT;
```

Remarks If *arg* is less than 32 bits, code is generated to extend the argument to 32 bits without sign extension.

Returns `BSR` right-shifts the bits of *arg* by the number of bits specified in *count*, modulo 64. Zeros are shifted into the high-order bit.

See also `BRotL`, `BRotR`, `BSL`, Listing 4.3.

BTst

Description Performs a bit test operation.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
FUNCTION BTst(
  arg    : ScalarType;
  bitNbr : IntegerType) : BOOLEAN;
```

Remarks If *arg* is less than 32 bits, code is generated to extend the argument to 32 bits without sign extension. The value of the bit to set, *bitNbr*, is reduced modulo 32. Bit 0 is considered the lowest-order bit, bit 31 the highest-order bit.

Returns `BTst` returns `TRUE` if the specified bit has the value one and returns `FALSE` if it has the value zero.

See also `BClr`, `BSet`

Listing 4.4 BTst example

```
PROGRAM BTst_example(input, output);

VAR
  val : LONGINT;
  bitcount : INTEGER;

BEGIN
  val := 43690;

  { Output bits, one at a time }
  FOR bitcount := 0 to 15 DO
    WriteLn(BTst(val, bitcount));
END.

(* Output:

FALSE
 TRUE
FALSE
 TRUE
FALSE
 TRUE
FALSE
 TRUE
FALSE
 TRUE
FALSE
 TRUE
FALSE
 TRUE
FALSE
 TRUE

*)
```

BXOR

Description Performs a bit-wise logical exclusive-OR operation.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
FUNCTION BXOR(
  arg1,
  arg2 : ScalarType) : LONGINT;
```

Remarks If *arg1* or *arg2* is less than 32 bits, code is generated to extend the arguments to 32 bits without sign extension.

Returns `BXOR` returns the logical XOR (exclusive-OR) of its two arguments.

See also `BAND`, `BNOT`, `BOR`, Listing 4.1.

Hi

Description Returns the higher 16 bits of a 32-bit value.

Compatibility Turbo Pascal

Prototype

```
FUNCTION Hi(
  arg : INTEGER | UNSIGNEDWORD) : UNSIGNEDBYTE
```

Returns `Hi` returns the highest-order byte of *arg*.

See also `HiWrd`, `LoWrd`

HiWrd

Description Returns the higher 16 bits of a 32-bit value.

Compatibility ANS Pascal, MPW Pascal

Prototype `FUNCTION HiWrd(arg : ScalarType) : INTEGER`

Remarks If *arg* is less than 32 bits, code is generated to extend the argument to 32 bits without sign extension. When *arg* is a simple variable or array access, no code is generated by this function because the argument is addressed and used as an integer.

Returns `HiWrd` returns the higher 16 bits of *arg*. If *arg* is not a `LONGINT`, `HiWrd` returns zero because of zero padding.

See also `LoWrd`

Listing 4.5 HiWrd example

```
PROGRAM HiWrd_Example(input, output);

CONST
  HigherValue = $1234;
  LowerValue  = $1235;
  BothValues  = $12341235;

BEGIN
  WriteLn(HigherValue);
  WriteLn(LowerValue);
  WriteLn;
  WriteLn(HiWrd(Value));
  WriteLn(LoWrd(Value));
END. { HiWrd_Example }

(* Output:

4660
4661

4660
4661

*)
```

Lo

Description Returns the lower 16 bits of a 32-bit value.

Compatibility Turbo Pascal

Prototype

```
FUNCTION Lo(
  arg : INTEGER | UNSIGNEDWORD) : UNSIGNEDBYTE
```

Returns `Lo` returns the lowest-order byte of *arg*.

See also `Hi`, Listing 4.5.

LoWrd

Description Returns the lower 16 bits of a 32-bit value.

Compatibility ANS Pascal, MPW Pascal

Prototype

```
FUNCTION LoWrd(arg : ScalarType) : INTEGER
```

Remarks If *arg* is less than 32 bits, code is generated to extend the argument to 32 bits without sign extension.

Returns `LoWrd` returns the lower 16 bits of *arg*.

See also `HiWrd`, Listing 4.5.

Swap

Description Exchanges the high and low order bytes in a variable .

Compatibility Turbo Pascal

Prototype

```
FUNCTION Swap(
  arg : IntegerType | UnsignedIntegerType) :
 IntegerType | UnsignedIntegerType
```

Returns `Swap` returns the value or arg with its high and low order bytes exchanged.

See also `HiWrd`, Listing 4.5.

5

Byte & Array Manipulation

Byte manipulation routines treat variables as raw sequences of bytes, without regard to their data type, to fill, move, and search through memory.

The packed character array routines operate only on arrays of type `PACKED ARRAY OF CHAR`.

Byte and Array Manipulation Routines Overview

The byte and array manipulation routines are:

`FillChar`	Stores the same value in a range of memory.
`High`	Counts the number of elements in an array.
`Move`	Copies memory.
`MoveLeft` `MoveRight`	Copies memory.
`Pack`	Converts a regular array to a packed array.
`ScanEq` `ScanNE`	Searches a range of memory.
`Unpack`	Converts a packed array to an unpacked array.

Unless otherwise specified these routines do no type checking on their *source* or *dest* actual parameters. However, because these are variable parameters, they cannot be indexed if they are packed since the index of a packed array does not match its corresponding offset in memory.

If an unpacked “byte array” is desired, then a variable of the type

```
ARRAY [lo..hi] OF –128..127
```

should be used for *source* or *dest*. The elements in an array of this type are stored in contiguous bytes; because it is unpacked, an array of this type can be used with an index as an actual parameter for these routines.

NOTE: An **unpacked** array with elements of the type `0 .. 255` or the type `CHAR` has its elements stored in 16-bit words, not bytes.

The packed character array routines operate only on arrays of type `PACKED ARRAY OF CHAR`. When used as parameters, such arrays cannot be subscripted; the routines described below always begin at their first character.

Byte and Array Manipulation Routines

FillChar

Description Fills a packed array of characters with a character.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype For ANS Pascal, MPW Pascal:

```
PROCEDURE FillChar(
  VAR paoc  : PackedCharArrayType;
  count : IntegerType;
  ch    : CHAR);
```

For Turbo Pascal:

```
PROCEDURE FillChar(
  VAR paoc;
  count : UnsignedInteger;
  ch);
```

Remarks The `FillChar` procedure fills a specified number of characters in a `PACKED ARRAY OF CHAR` with a specified character.

For the ANS Pascal, MPW Pascal I/O models *paoc* is an expression with a value of type `PACKED ARRAY OF CHAR`. It is a variable parameter. For the Turbo Pascal I/O model, *paoc* is a typeless variable parameter that is treated by `FillChar` as a `PACKED ARRAY OF CHAR`.

The *count* parameter is truncated to 16 bits and is not range-checked.

For the ANS Pascal, MPW Pascal I/O models *ch* is an expression with a value of type `CHAR`. For the Turbo Pascal I/O model, *ch* is a typeless variable parameter that is treated by `FillChar` as a `CHAR`.

`FillChar` writes the value of *ch* into *count* contiguous bytes of memory, starting at the first byte of *paoc*. Because the *count* parameter is not range checked, it is possible to write into memory outside of *paoc*, with unspecified results.

See also `ScanEq`, `ScanNE`

Listing 5.1 FillChar, ScanEq, and ScanEQ example

```
PROGRAM FillChar_Example(input, output);

CONST
  Max = 1000;
  Blank = ' ';
  Bang = '!';

VAR
  a : PACKED ARRAY [1 .. Max] OF CHAR;

BEGIN
  FillChar(a, Max, Blank);
  a[500] := Bang;
  WriteLn(ScanEq(Max, Bang, a));
  WriteLn(ScanNE(Max, Blank, a));
END. { FillChar }
```

```
(* Output:

499
499

*)
```

High

Description Returns the number of elements in an array.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
FUNCTION High(
  VAR a : OpenArrayParameterType ) : INTEGER;
```

Remarks The *a* parameter must be passed to the currently executing routine as an open array. The *a* parameter cannot be called with a local or global variable.

Returns `High` returns the number of elements in an array, counting the first element as 0.

Listing 5.2 High example

```
PROGRAM High_Example(input, output);

PROCEDURE ClearArray(VAR a : ARRAY OF INTEGER);
VAR
  i : INTEGER;
BEGIN
  FOR i := 0 to High(a) DO
    a[i] := 0;
END; (* ClearArray *)

VAR
  count : INTEGER;
  myArray : ARRAY [1 .. 200] OF INTEGER;
```

```
BEGIN
  FOR count := 1 TO 200 DO
    myArray[count] := 1;
  ClearArray(myArray);
  FOR count := 1 TO 200 DO
    IF myArray[count] <> 0 THEN
      WriteLn('Element ', count, ' is not zero!');
END.

(* Output:

*)
```

Move

Description Moves memory starting at the "left."

Compatibility Turbo Pascal

Prototype

```
PROCEDURE Move(
  VAR source,
  dest;
  count       : UnsignedIntegerType);
```

Remarks The `Move` procedure copies count contiguous bytes from the address of source to the address of dest, starting at the lowest address.

Source is a variable reference that refers to a variable of any type except a file type or a structured type that contains a file type. It is a variable parameter. The first byte allocated to *source* (lowest address within *source*) is the first byte of the source range.

Dest is a variable reference that refers to a variable of any type except a file type or a structured type that contains a `FILE` type. It is a variable parameter. The first byte allocated to *dest* (lowest address within *dest*) is the first byte of the destination range.

Count is an expression with an `unsigned integer` value. The source range and the destination range are each *count* bytes long. The *count* parameter is not range checked.

See also `MoveLeft`, `MoveRight`

MoveLeft

Description Moves memory starting at the "left."

Compatibility ANS Pascal, MPW Pascal

Prototype

```
PROCEDURE MoveLeft(
  VAR source,
  dest        : NonFILEType;
  count       : INTEGER);
```

Remarks The `MoveLeft` procedure copies a specified number of contiguous bytes from a source range to a destination range (starting at the lowest address).

Source is a variable reference that refers to a variable of any type except a file type or a structured type that contains a file type. It is a variable parameter. The first byte allocated to *source* (lowest address within *source*) is the first byte of the source range.

Dest is a variable reference that refers to a variable of any type except a file type or a structured type that contains a `FILE` type. It is a variable parameter. The first byte allocated to *dest* (lowest address within *dest*) is the first byte of the destination range.

Count is an expression with an `INTEGER` value. The source range and the destination range are each *count* bytes long. The *count* parameter is not range checked.

See also `MoveRight`

Listing 5.3 MoveLeft and MoveRight example

```
PROGRAM Move_Example(input, output);

CONST
  CharSize = SizeOf(CHAR);
  ArraySize = 10;

VAR
  myArray : PACKED ARRAY [1 .. ArraySize] OF CHAR;
  i : INTEGER;

BEGIN
  FOR i := 1 TO ArraySize DO
    myArray[i] := Chr(Ord('A') + i - 1);

  WriteLn('Before:');
  FOR i := 1 TO ArraySize DO
    Write(myArray[i]:3);
  WriteLn;

  MoveLeft(@myArray[1], @myArray[3], CharSize * 4);
  WriteLn('MoveLeft:');
  FOR i := 1 TO ArraySize DO
    Write(myArray[i]:3);
  WriteLn;

  FOR i := 1 TO ArraySize DO
    myArray[i] := Chr(Ord('A') + i - 1);

  MoveRight(@myArray[1], @myArray[3], CharSize * 4);
  WriteLn('MoveRight:');
  FOR i := 1 TO ArraySize DO
    Write(myArray[i]:3);
  WriteLn;
END. { Move_Example }

(* Output:

Before:
```

```
  A  B  C  D  E  F  G  H  I  J
MoveLeft:
  A  B  A  B  C  D  E  F  G  H
MoveRight:
  A  B  A  B  C  D  E  F  G  H

*)
```

MoveRight

Description Moves memory starting at the "right."

Compatibility ANS Pascal, MPW Pascal

Prototype

```
PROCEDURE MoveRight(
  VAR source,
  dest       : NonFILEType;
  count      : INTEGER);
```

Remarks `MoveRight` is exactly like `MoveLeft`, except that it starts from the "right" end of the source range (highest address). It proceeds to the "left" (lower addresses), copying bytes into the destination range, starting at the highest address of the destination range.

The reason for having both `Moveleft` and `Moveright` is that the source and destination ranges may overlap. If they overlap, the order in which bytes are moved is critical: each byte must be moved before it gets overwritten by another byte.

See also `MoveLeft`, Listing 5.3.

Pack

Description Copies elements from a regular array to a packed array.

Compatibility **ANS Conformance** in the **Pascal Language** settings panel must be turned on.

Prototype

```
PROCEDURE Unpack(
  VAR source  : ARRAY [j .. k] OF T;
  index       : IntegerType;
  VAR dest    : PACKED ARRAY [m .. n] OF T);
```

Remarks `Pack` copies the elements of array *source* into the elements into packed array *dest*, starting at element *source*`[`*index*`]`. Both *source* and *dest* arrays must have elements of the same type, *T*.

See also Unpack

ScanEq

Description Looks for a character in a packed array of characters.

Compatibility ANS Pascal, MPW Pascal

Prototype

```
FUNCTION ScanEq(
  lim  : IntegerType;
  ch   : CHAR;
  paoc : PackedCharArrayType) : INTEGER;
```

Remarks `ScanEq` scans *paoc*, looking for the first occurrence of *ch*. The scan begins with the first character in *paoc*. The scan proceeds sequentially through *paoc* and ends when *lim* elements from the first element in *paoc* have been scanned.

Lim is an expression with a value of type `INTEGER` or `LONGINT`. It is truncated to 16 bits and is not range checked. *Lim* specifies the last element to in *paoc* to scan.

Ch is an expression with a value of type `CHAR`.

Paoc is an expression with a value of type `PACKED ARRAY OF CHAR`.

Returns `ScanEq` returns an `INTEGER` value equal to *lim* if the character *ch* is not found within *lim* characters from the beginning of *paoc*. Otherwise, the value returned is an `INTEGER` that gives the number of characters scanned before *ch* was found.

See also Pos, ScanNE, Listing 5.1.

ScanNE

Description Looks for a character that is not in a packed array of characters.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
FUNCTION ScanNE(
  lim  : IntegerType;
  ch   : CHAR;
  paoc : PackedCharArrayType) : INTEGER;
```

Remarks ScanNE scans *paoc*, looking for the first occurrence of a character that is not *ch*. The scan begins with the first character in *paoc*. The scan proceeds sequentially through *paoc* and ends when *lim* elements from the first element in *paoc* have been scanned.

Lim is an expression with a value of type INTEGER or LONGINT. It is truncated to 16 bits and is not range checked. *Lim* specifies the last element to in *paoc* to scan.

Ch is an expression with a value of type CHAR.

Paoc is an expression with a value of type PACKED ARRAY OF CHAR.

Returns ScanNE returns an INTEGER value equal to *lim* if all characters scanned in *paoc* have the value *ch*. Otherwise, the value returned is an INTEGER that gives the number of characters scanned before a character that is not *ch* was found.

See also ScanEq, Pos, Listing 5.1.

Unpack

Description Copies elements from a packed array to an unpacked array.

Compatibility **ANS Conformance** in the **Pascal Language** settings panel must be turned on.

Prototype

```
PROCEDURE Pack(
  VAR source  : PACKED ARRAY [j .. k] OF T;
  VAR dest    : ARRAY [m .. n] OF T;
  index       : IntegerType);
```

Remarks `Unpack` copies the elements of packed array *source* into the elements into array *dest*, starting at element *dest*`[`*index*`]`. Both *source* and *dest* arrays must have elements of the same type, *T*.

See also `Pack`

6

Exit and Halt Routines

These routines terminate program execution unconditionally.

Exit and Halt Routines Overview

The exit and halt routines are:

`Exit`	Leaves a statement block.
`Halt`	Stops the program.
`RunError`	Stops the program and returns an error code to the operating system.

Exit and Halt Routines

Exit

Description Exits from a procedure, function, or main program.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype For ANS Pascal, MPW Pascal:

```
PROCEDURE Exit [(
  id : ProceduralType | PROGRAM) ];
```

For Turbo Pascal:

```
PROCEDURE Exit;
```

Remarks The `Exit` procedure exits immediately from a specified procedure or function or from the main program.

The parameter *id* is the identifier of a procedure or function, or of the main program, in the scope of the `Exit` call. You can also use the reserved word `PROGRAM` to identify the currently executing program.

The call `Exit`(*id*) causes an immediate exit from *id*. Essentially, it causes a jump to the end of *id*. The routine identified by *id* must be part of the current dynamic calling chain. Using the name of the program is illegal.

The Turbo Pascal I/O model allows an additional variation. The call

```
Exit;
```

exits the currently executing routine.

See also `Halt`

Listing 6.1 Exit and Halt example

```
PROGRAM Exit_Example(input, output);

VAR
  x, y : REAL;

PROCEDURE CheckForZero(theValue : REAL);
BEGIN
  IF theValue = 0 THEN
    Exit(CheckForZero);
  WriteLn('The divisor is not zero.');
END; { CheckForZero }

BEGIN
  REPEAT
    WriteLn;
    Write('Enter two numbers: ');
```

```
    ReadLn(x, y);
    CheckForZero(y);
    IF y = 0 THEN
      Halt { Can't divide by zero, leave }
    ELSE
      WriteLn('x / y = ', x / y);
  UNTIL FALSE; { Loop forever }
END. { Exit_Example }

(* Output:

Enter two numbers: 20.0 4.0
The divisor is not zero.
x / y =  5.0e+0000

Enter two numbers: 25.0 0

*)
```

Halt

Description Exits from the main program.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype For ANS Pascal, MPW Pascal:

```
PROCEDURE Halt;
```

For Turbo Pascal:

```
PROCEDURE Halt [(
  exitCode : UnsignedIntegerType ) ];
```

Remarks `Halt` with no parameter causes an immediate exit from the main program.

`Halt` with a parameter causes an immediate exit from the main program and returns *exitCode* to the operating system. Note that the operating system won't necessarily act on the *exitCode* value passed to

it by `Halt`. Consult your operating system's documentation for details.

See also `Exit`, Listing 6.1

RunError

Description Exits from the main program and returns an error code to the operating system.

Compatibility Turbo Pascal

Prototype

```
PROCEDURE RunError [(errorCode : ByteValueType ) ];
```

Remarks Calling `RunError` causes an immediate exit from the main program and returns *errorCode* to the operating system. Note that the operating system won't necessarily act on the *exitCode* value passed to it by `RunError`. Consult your operating system's documentation for details.

See also `Halt`

7

Input and Output

Consult the Pascal Language Manual for information on input/ output.

Input/Output Routines Overview

CodeWarrior Pascal provides input/output routines to read and write files and console input/output.

The routines in this chapter are organized into Compatibility. Each model specifies which routines are available and the parameters they accept at compile time, and what their behaviors are at runtime.

For more information on Compatibility, see "Choosing Which I/O Routines to Use" on page 73. For information on which I/O model a routine is available under, see that routine's topic.

This section organizes input/output routines into these categories:

- Routines for All Files
- Routines for Structured and Untyped Files
- Routines for Structured and Text Files
- Routines for Raw Data File Access
- Routines for Text File Access
- Choosing Which I/O Routines to Use

Routines for All Files

These routines are used on text files (`TEXT`), structured files (`FILE OF` *DataType*), and untyped files (`FILE`).

Assign	Associates a file name with a file variable.
Close	Closes an open file.
EOF	Returns a file's end-of-file status.
Erase	Deletes a file.
IOResult, InOutRes	Returns the status of the last I/O operation. Note that InOutRes is a variable.
Open	Opens a file for reading or creates a new file for writing.
PLCrunch	Truncates a file.
PLPurge	Deletes a file.
PLRename	Renames a file.
Rename	Renames a file.
Reset	Opens a file for reading.
Rewrite	Opens a file for writing.

Routines for Structured and Untyped Files

These routines provide random and sequential access to structured files (`FILE OF` *DataType*) and untyped files (`FILE`):

FileMode	Sets how files are accessed with the Reset routine.
FilePos	Returns the current file position.
FileSize	Returns the size of a file.
InOutRes, IOResult	Returns the status of the last I/O operation.
Seek	Jumps to an item within a file.
Truncate	Removes data after the current file position.

Routines for Structured and Text Files

These routines provide access to structured files (`FILE OF` *DataType*) and text files (`TEXT`):

`Get`	Reads an item.
`Put`	Writes an item.
`Read`	Reads an item.
`Write`	Writes an item.

Routines for Raw Data File Access

These routines allow block-oriented raw-data access to files:

`BlockRead`	Reads a fixed-size chunk of data.
`BlockWrite`	Writes a fixed-size chunk of data.
`ByteRead`	Reads a byte.
`ByteWrite`	Writes a byte.

Routines for Text File Access

These routines are used on text-oriented files.

`EOLn`	Returns a text file's end-of-line status.
`Get`	Reads a record.
`Page`	Outputs a form feed.
`Put`	Writes a record.
`PLFlush`	Ensures that a buffer is written to a file.
`PLSetVBuf`	Specifies buffer management.
`Read`	Reads a piece of formatted text.
`ReadLn`	Reads a line of formatted text.
`SeekEOF`	Skips white space characters before checking for end-of-file.

SeekEOLn	Skips white space characters before checking for end-of-line.
SetTextBuf	Assigns a buffer for a text file.
Write	Formats then writes a piece of text.
WriteLn	Formats then writes a line of text.

Files of type TEXT are distinguished from other kinds of files (for example, FILE OF CHAR) by the special significance given to the end-of-line character.

The end-of-line character allows a file of type TEXT to be treated as a sequence of lines, rather than a sequence of individual characters. An entire line may be read from the file into a STRING type (note that STRING variables are limited to 255 characters, the maximum size for this data type) or PACKED ARRAY OF CHAR variable using the ReadLn procedure, and an entire line may be written to the file by using the WriteLn procedure. You can test for the end-of-line character by using the EOLn function described later in this document.

When the value of the logical record at the current file position of a file is an end-of-line character, reading a character with the Read and ReadLn procedures reads as a space character (ASCII 32). In the Turbo Pascal I/O Model, if end-of-file is true, reading a character produces an EOF character (ASCII 26).

The WriteLn and Write procedures can be applied to any typed file. However, the procedures ReadLn and WriteLn depend upon the presence of the end-of-line character, which appears only in files of type TEXT.

None of the text-oriented predefined procedures and functions need an explicit file variable parameter. If no file is named, one of the predefined files, input or output, will be assumed by default, depending on whether the procedure or function is input-oriented or output-oriented. Remember that input and output are predeclared as files of type TEXT.

The BlockRead, BlockWrite, ByteRead, and ByteWrite routines are used on untyped files—that is, variables of type FILE with no specified logical record type. An untyped file has no file window variable, and it cannot be used with the Get or Put procedure or with any of the text-oriented I/O procedures. It can only be used with Reset, Rewrite, Open, Close, Seek, EOF, and BlockRead, BlockWrite, ByteRead, ByteWrite.

With BlockRead and BlockWrite, an untyped file is treated as a sequence of fixed-size blocks. The size of the blocks depends on the I/O model. With ByteRead and ByteWrite, it is treated as a sequence of bytes. In both cases, the file bytes are not type-checked but are considered raw data. This can be useful for applications where the data need not be interpreted at all during I/O operations.

The blocks in an untyped file are considered to be numbered sequentially starting with logical record 0. The system keeps track of the current block number; it is block 0 immediately after the file is opened. Each time a block is read, the current block number is incremented. By default, each I/O operation begins at the current block number; however, an arbitrary block number can be specified.

To use untyped file I/O, an untyped file is opened with Open, Reset, or Rewrite. The BlockRead, BlockWrite, ByteRead, and ByteWrite functions may then be used for input and output.

Choosing Which I/O Routines to Use

CodeWarrior Pascal supports several input/output *models* for compatibility with other implementations of Pascal. Each model provides a set of I/O routines to make it easier to compile source code written with other Pascal compilers.

To set the I/O model used in your CodeWarrior Pascal project, use the **IO Model** pop-up menu in the Pascal Language Settings.

For information on the availability and behavior of specific I/O routines, refer to that routine's topic in this chapter.

The I/O models are:

- ANS Pascal
- MPW Pascal
- Turbo Pascal

For information on the availability of I/O routines for each model, see Table 7.1.

Table 7.1 Input/output routine availability for each I/O model

Routine/variable	ANS I/O Model	MPW I/O Model	Turbo I/O Model
`Append`	not available	not available	for text files
`Assign`	not available	not available	for all files
`BlockRead`	for untyped files	for untyped files	for untyped files
`BlockWrite`	for untyped files	for untyped files	for untyped files
`ByteRead`	for untyped files	for untyped files	not available
`ByteWrite`	for untyped files	for untyped files	not available
`Close`	for all files	for all files	for all files
`EOF`	for all files	for all files	for all files
`EOLn`	for text files	for text files	for text files
`Erase`	not available	not available	for all files
`f^`	for all files	for all files	not available
`FileMode`	not available	not available	for structured and untyped files
`FilePos`	not available	not available	for structured and untyped files
`FileSize`	not available	not available	for structured and untyped files
`Flush`	not available	not available	for text files
`Get`	for text and structured files	for text and structured files	not available

Routine/variable	ANS I/O Model	MPW I/O Model	Turbo I/O Model
InOutRes	not available	not available	for all files
IOResult	for all files	for all files	for all files
Open	for all files	for all files	not available
Page	for text files	for text files	not available
PLCrunch	not available	for all files	not available
PLFilePos	not available	for all files	not available
PLFlush	not available	for text files	not available
PLPurge	not available	for all files	not available
PLRename	not available	for all files	not available
PLSetVBuf	not available	for text files	not available
Put	for text and structured files	for text and structured files	not available
Read	for text and structured files	for text and structured files	for text and structured files
ReadLn	for text files	for text files	for text files
Reset	for all files	for all files	for all files
Rename	not available	not available	for all files
Rewrite	for all files	for all files	for all files
Seek	for all files	for all files	for all files
SeekEOF	not available	not available	for text files
SeekEOLn	not available	not available	for text files
SetTextBuf	not available	not available	for text files
Truncate	not available	not available	for all files

Routine/variable	ANS I/O Model	MPW I/O Model	Turbo I/O Model
Write	for text and structured files	for text and structured files	for text and structured files
WriteLn	for text files	for text files	for text files

Input and Output Routines

Append

Description Opens a file so that data may be added to it.

Compatibility Turbo Pascal

Prototype

```
PROCEDURE Append(
  VAR f           : TEXT);
```

Remarks The Append procedure opens an existing file in appending mode. That is, the file is opened in write-only mode and the file cursor is moved to the end of the file. If a Control-Z character (ASCII 26) is found, the file cursor is set to overwrite it. If the file is already open, it is closed then reopened.

Parameter *f* is a text file. It must refer to a file that already exists. If *f* refers to standard input or output, it is reassigned to standard output.

See also Rewrite

Assign

Description Assigns a file name to a file pointer.

Compatibility Turbo Pascal

Prototype

```
PROCEDURE Assign(
  VAR f       : FileType
```

```
    name        : STRING);
```

Remarks The Assign routine assigns a file name, *name*, to a file, *f*. If the file name is empty, then the standard input/output file is assigned to *f*. If the file being assigned to *f* is already open, it is first closed before being reopened.

Parameter *f* may be a text file, structured, or untyped file.

Parameter *name* is a string of at most 79 characters. If it is longer than 79 characters, it is truncated.

See also `Reset`, `Rewrite`

BlockRead

Description Reads blocks of data from an untyped file.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype ANS Pascal and MPW Pascal

```
FUNCTION BlockRead(
  VAR f      : FILE;
  databuf    : RefType;
  count      : IntegerType [;
  blocknum   : IntegerType]): INTEGER;
```

Prototype Turbo Pascal

```
PROCEDURE BlockRead(
  VAR f       : FILE;
  databuf     : RefType;
  count       : IntegerType [;
  VAR result  : UnsignedIntegerType]);
```

Remarks ANS Pascal **and** MPW Pascal

The `BlockRead` function reads one or more 512-byte blocks of data from an untyped file to a program variable and returns an `INTEGER` representing the number of blocks read.

The parameter *f* is a variable reference that refers to a variable of type `FILE`. The file must be open.

Databuf is a variable reference that refers to the variable into which the blocks of data will be read. The size and type of this variable are not checked; if it is not large enough to hold the data, other data are overwritten and the results are unpredictable.

Count is an expression with an `INTEGER` value. It specifies the maximum number of blocks to be transferred. `BlockRead` will read blocks until this limit is reached, the end-of-file is reached, or an error occurs.

Blocknum is an optional expression with an `INTEGER` value. It specifies the starting block number for the transfer. If it is omitted, the transfer begins with the current block. Thus the transfers are sequential if the *blocknum* parameter is never used; if a *blocknum* parameter is used, it provides random access to blocks.

After the last block in the file has been read, the current block number is unspecified and `EOF`(*f*) is `TRUE`. Otherwise, `EOF`(*f*) is `FALSE` and the current block number is advanced to the block after the last block that was read. If the end of the file occurs while the last block is being read, the remainder of the block is filled with zero bytes.

Remarks Turbo Pascal

The `BlockRead` function reads one or more blocks of data from an untyped file to a program variable. By default the size of the blocks is 128 bytes. The size of the blocks may be optionally set when the file is opened using the `Reset` and `Rewrite` routines.

The parameter *f* is a variable reference that refers to a variable of type `FILE`. The file must be open.

Databuf is a variable reference that refers to the variable into which the blocks of data will be read. The size and type of this variable are not checked; if it is not large enough to hold the data, other data are overwritten and the results are unpredictable.

Count is an expression with an `INTEGER` value. It specifies the maximum number of blocks to be transferred. `BlockRead` will read blocks until this limit is reached, the end-of-file is reached, or an error occurs.

Result is an optional variable parameter in which is stored the number of blocks actually read. If this parameter isn't used and less than *count* blocks are read, an I/O error occurs.

After the last block in the file has been read, the current block number is unspecified and `EOF`(*f*) is `TRUE`. Otherwise, `EOF`(*f*) is `FALSE` and the current block number is advanced to the block after the last block that was read. If the end of the file occurs while the last block is being read, the remainder of the block is filled with zero bytes.

Returns **ANS Pascal and MPW Pascal**

`BlockRead` returns the number of blocks read.

If `BlockRead` reads fewer than *blocknum* blocks, it returns the actual number of blocks read. If `EOF`(*f*) is `TRUE` when `BlockRead` is called, `BlockRead` returns zero and `IOResult` returns a value of `eofErr` (–39).

Returns **Turbo Pascal**

`BlockRead` optionally returns the number of blocks read in its *result* parameter.

See also `BlockWrite`, `ByteRead`, `IOResult`

Listing 7.1 BlockRead and BlockWrite example

```
PROGRAM BlockRead_Example(input, output);

(* For the ANS Pascal and the MPW Pascal I/O models. *)

CONST
  BufferSize = 512;
  TestCharacter = '*';
```

```
VAR
  i : INTEGER;
  f : FILE;
  fileName : STRING[255];

  myBuffer : PACKED ARRAY [1 .. BufferSize] OF CHAR;

BEGIN
  FOR i := 1 TO BufferSize DO
    myBuffer[i] := TestCharacter;

  Write('Enter the name of a file to create: ');
  ReadLn(fileName);

  { Create a new file }
  Rewrite(f, fileName);
  IF BlockWrite(f, myBuffer, 1) <> 1 THEN
  BEGIN
    WriteLn('Error writing ', fileName);
    Close(f);
    Exit(PROGRAM);
  END;
  Close(f);

  { Clear myBuffer }
  FOR i := 1 TO BufferSize DO
    myBuffer[i] := ' ';

  { Read the newly created file }
  Open(f, fileName);
  IF BlockRead(f, myBuffer, 1) <> 1 THEN
  BEGIN
    WriteLn('Error reading ', fileName);
    Close(f);
    Exit(PROGRAM);
  END;
  Close(f);
  WriteLn('Done.');
END.
```

```
(* Output:

Enter the name of a file to create: blocktest
Done.

*)
```

BlockWrite

Description Writes blocks of data to an untyped file.

Compatibility ANS Pascal, MPW Pascal, and Turbo Pascal

Prototype ANS Pascal and MPW Pascal

```
FUNCTION BlockWrite(
  VAR f       : FILE;
  databuf     : RefType;
  count       : IntegerType [;
  blocknum    : IntegerType]): INTEGER;
```

Prototype Turbo Pascal

```
PROCEDURE BlockWrite(
  VAR f       : FILE;
  databuf     : RefType;
  count       : IntegerType [;
  VAR result  : UnsignedIntegerType]);
```

Remarks ANS Pascal **and** MPW Pascal

The `BlockWrite` function writes one or more 512-byte blocks of data from a buffer to an untyped file and returns an `INTEGER` representing the number of blocks written.

The parameter *f* is a variable reference that refers to a variable of type `FILE`. The file must be open.

Databuf is a variable reference that refers to the variable into which the blocks of data will be written. The size and type of this variable are not checked.

Count is an expression with an `INTEGER` value. It specifies the maximum number of blocks to be transferred. `BlockWrite` will write blocks up to this limit unless an error occurs.

Blocknum is an optional expression with an `INTEGER` value. It specifies the starting block number for the transfer. If it is omitted, the transfer begins with the current block. Thus the transfers are sequential if the *blocknum* parameter is never used; if a *blocknum* parameter is used, it provides random access to blocks.

If disk space runs out during the data transfer, the current block number is unspecified. Otherwise, the current block number is advanced to the block after the last block that was written.

Remarks — Turbo Pascal

The `BlockWrite` function writes one or more blocks of data from a buffer to an untyped file. By default the size of the blocks is 128 bytes. The size of the blocks may be optionally set when the file is opened using the `Reset` and `Rewrite` routines.

The parameter *f* is a variable reference that refers to a variable of type `FILE`. The file must be open.

Databuf is a variable reference that refers to the variable from which the blocks of data will be written. The size and type of this variable are not checked.

Count is an expression with an `INTEGER` value. It specifies the maximum number of blocks to be transferred. `BlockWrite` will write blocks up to this limit unless an error occurs.

Result is an optional variable parameter in which is stored the number of blocks actually written. If this parameter isn't used and less than *count* blocks are written, an I/O error occurs.

If disk space runs out during the data transfer, the current block number is unspecified. Otherwise, the current block number is advanced to the block after the last block that was written.

Returns

ANS Pascal and MPW Pascal

`BlockWrite` returns the actual number of blocks written.

Turbo Pascal

`BlockWrite` optionally returns the number of blocks written in its *result* parameter.

See also `BlockRead`, `ByteRead`, `IOResult`, Listing 7.1

ByteRead

Description Transfers bytes from an untyped file.

Compatibility ANS Pascal, MPW Pascal

Prototype

```
FUNCTION ByteRead(
  VAR f        : FILE;
  databuf      : RefType;
  count        : LONGINT [;
  bytenum      : LONGINT]): LONGINT;
```

Remarks The `ByteRead` function perform identically to `BlockRead` except for a few differences.

- `ByteRead` transfers bytes of data instead of blocks.
- The parameters *bytenum* and *count* and the function result have a data type of `LONGINT` instead of `INTEGER`.
- The value *bytenum* is the current byte position in the file. The first byte is numbered zero.
- `ByteRead` returns the number of bytes transferred.

NOTE: Mixing block and byte untyped file functions can result in confusion unless their blocknum and bytenum parameters are

used to adjust the current file position. A block function always transfers the next 512 bytes. After a byte function, though, this may no longer conform to a natural block boundary.

Returns `ByteRead` returns the number of bytes actually read.

See also `BlockRead`, `BlockWrite`, `ByteWrite`.

Listing 7.2 ByteRead and ByteWrite example

```
PROGRAM ByteRead_example(input, output);

(* For the ANS Pascal or the MPW Pascal I/O models. *)

CONST
  BufferSize = 512;
  TestCharacter = '*';

VAR
  i : INTEGER;
  f : FILE;
  fileName : STRING[255];
  myBuffer : PACKED ARRAY [1..BufferSize] OF CHAR;

BEGIN

  FOR i := 1 TO BufferSize DO
    myBuffer[i] := TestCharacter;

  Write('Enter the name of a file to create: ');
  ReadLn(fileName);

  { Create a new, empty file }
  Rewrite(f, fileName);
  IF ByteWrite(f, myBuffer, BufferSize)
      <> BufferSize THEN
    WriteLn('Error writing ', fileName);
    Close(f);
    Exit(PROGRAM);
```

```
  Close(f);

  { Clear myBuffer }
  FOR i := 1 TO BufferSize DO
    myBuffer[i] := ' ';

  Open(f, fileName);
  IF ByteRead(f, myBuffer, BufferSize)
      <> BufferSize THEN
    WriteLn('Error reading ', fileName);
    Close(f);
    Exit(PROGRAM);
  Close(f);
  WriteLn('Done.');
END.

(* Output:

Enter the name of a file to create: bytetest
Done.

*)
```

ByteWrite

Description Transfers bytes to an untyped file.

Compatibility ANS Pascal, MPW Pascal

Prototype

```
FUNCTION ByteWrite(
  VAR f        : FILE;
  databuf      : RefType;
  count        : LONGINT [;
  bytenum      : LONGINT]): LONGINT;
```

Remarks The `ByteWrite` function performs identically to `BlockWrite` except for a few differences.

- `ByteWrite` transfers bytes of data instead of blocks.

- The parameters *bytenum* and *count* and the function result have a data type of `LONGINT` instead of `INTEGER`.
- The value *bytenum* is the current byte position in the file. The first byte is numbered zero.
- `ByteWrite` returns the number of bytes transferred.

NOTE: Mixing block and byte untyped file functions can result in confusion unless their blocknum and bytenum parameters are used to adjust the current file position. A block function always transfers the next 512 bytes. After a byte function, though, this may no longer conform to a natural block boundary.

Returns `ByteWrite` returns the number of bytes actually written.

See also `BlockRead`, `BlockWrite`, `ByteRead`, Listing 7.2.

Close

Description Closes an open file.

Compatibility ANS Pascal, MPW Pascal, and Turbo Pascal

Prototype

```
PROCEDURE Close(VAR f : FileType);
```

Remarks The `Close` procedure closes an open file. It ends the association between the file variable and the external file, if one exists.

The parameter *f* is a variable reference that refers to a file variable, which must be open.

The statement `Close(`*f*`)` closes *f*. That is, the association between *f* and its external file is broken, and the file system marks the external file closed. All subsequent references to *f* are invalid (except to open it again). In particular, the value of *f*^ becomes undefined.

If a file has not been closed during program execution, it is closed automatically when the program terminates.

`Close` can be used with files of all kinds, both typed and untyped.

WARNING! Files that have local scope in a procedure or function block are not automatically closed when the procedure or function is exited.

See also `Open`

Listing 7.3 Close and Rewrite example

```
PROGRAM Close_example(input, output);

(* For all I/O models. *)

CONST

VAR
  f : TEXT;
  fileName, textIn : STRING[255];
  count : INTEGER;

BEGIN
  Write('Enter a file name: ');
  ReadLn(fileName);

  { Create a new, empty file }
  Rewrite(f, fileName);

  { Write to the new, empty file }
  FOR count := 1 to 10 DO
    WriteLn(f, count);

  { Close the new file }
  Close(f);

  { Open the existing file }
  Open(f, fileName);
```

```
  { Send the file's contents to screen }
  WHILE NOT EOF(f) DO
  BEGIN
    ReadLn(f, textIn);
    WriteLn(textIn);
  END;

  { Close the existing file }
  Close(f);
END.

(* Output:

Enter a file name: test.txt
1
2
3
4
5
6
7
8
9
10

*)
```

EOF

Description Determines if the file position is at the end-of-file.

Compatibility ANS Pascal, MPW Pascal, and Turbo Pascal

Prototype

```
FUNCTION EOF[(VAR f : FileType)] : BOOLEAN;
```

Returns The `EOF` function returns a `boolean` value that indicates whether or not the current file position is the end of the file.

The parameter *f* is a variable reference that refers to a file variable. If *f* is omitted, the function is applied to the predefined file `input`. The file must be open, or an error occurs.

`EOF` can be used with files of all kinds, both typed and untyped.

- The `EOF` function returns `TRUE` if the file position is beyond the last logical record of the file
- The `EOF` function returns `TRUE` if the file contains no logical records
- The `EOF` function returns `TRUE` after a `Get` procedure, if the current file position is the last logical record of the file
- The `EOF` function returns `TRUE` after a `Put` procedure, if the logical record written by the `Put` is now the last logical record
- In all other cases, `EOF` returns `FALSE`.

See also `EOLn`, `IOResult`

Listing 7.4 EOF and Reset example

```
PROGRAM EOF_Example(input, output);

(* For all I/O models. *)

VAR
  fileName : STRING[100];
  f : TEXT;
  inString : STRING[1000];

BEGIN
  Write('Enter the name of a text file to read: ');
  ReadLn(fileName);
  Reset(f, fileName);
  WHILE NOT EOF(f) DO
  BEGIN
    ReadLn(f, inString);
    WriteLn(inString);
  END; { WHILE }
  Close(f);
```

```
END. { EOF_Example }

(* Output:

Enter the name of a text file to read: test.txt
CodeWarrior
Arnold

*)
```

EOLn

Description Determines if the file position is at the end-of-line.

Compatibility ANS Pascal, MPW Pascal, and Turbo Pascal

Prototype

```
FUNCTION EOLn[(VAR f : TEXT )] : BOOLEAN;
```

Remarks The parameter *f* is a variable reference that refers to a file variable of type `TEXT`. The file must be open. If *f* is omitted, the function is applied to the standard file `input`.

Returns `EOLn` returns `true` if the character at the current file position is an end-of-line character. An error occurs if `EOLn(`*f*`)` is applied to a non-text file or if *f* is write-only. If `EOF(`*f*`)` is `TRUE`, `EOLn(`*f*`)` returned `TRUE`.

See also `EOF`, `IOResult`

Listing 7.5 EOLn example

```
PROGRAM EOLn_Example(input, output);

(* For all I/O models. *)

VAR
  fileName : STRING[100];
  f : TEXT;
```

```
  inChar : CHAR;


BEGIN
  Write('Enter the name of a text file to read: ');
  ReadLn(fileName);
  Reset(f, fileName);
  WHILE NOT EOF(f) DO
  BEGIN
    IF EOLn(f) THEN
    BEGIN
      WriteLn;
      Read(f, inChar);
    END; { IF }
    Read(f, inChar);
    Write(inChar);
  END; { WHILE }
  IF EOLn(f) THEN { Check for last end-of-line }
  BEGIN
    WriteLn;
    Read(f, inChar);
  END; { IF }
  Close(f);
END. { EOLn_Example }

(* Output:

Enter the name of a text file to read: test.txt
CodeWarrior
Arnold

*)
```

Erase

Description Deletes a file.

Compatibility Turbo Pascal

Prototype

```
PROCEDURE Erase(VAR f : FileType);
```

Remarks `Erase` deletes the file referred to by *f*. The file must exist. If it is open, it is closed before it's deleted. Use `Assign` to associate *f* with a file.

See also `Assign`, `PLPurge`

FileMode

Description Specifies how files should be opened with the `Reset` routine.

Compatibility Turbo Pascal

Prototype

```
VAR
  FileMode : INTEGER;
```

Remarks The `FileMode` variable specifies how structured and untyped files opened with `Reset` are accessed. Modifying the `FileMode` variable affects subsequent calls to `Reset`.

`FileMode` doesn't affect text files.

Legal values for `FileMode` are:

- `0` for read only
- `1` for write only
- `2` for read and write

By default, `FileMode` is set to 2.

See also `Reset`, `Rewrite`

Listing 7.6 FileMode example

```
PROGRAM FileMode_Example(input, output);

(* For Turbo Pascal *)

VAR
```

```
  temp : INTEGER;
  f : FILE OF CHAR;

BEGIN
  (* This piece of code is equivalent to
   *     Rewrite(f, 'Foofile.dat');
   *)

  temp := FileMode; (* Save the old value *)
  FileMode := 2;
  Reset(f, 'Foofile.dat');
  FileMode := temp;
  (* ... *)
END.
```

FilePos

Description Returns the current file position.

Compatibility Turbo Pascal

Prototype

```
FUNCTION FilePos(
  VAR f : FileType) : LONGINT;
```

Remarks `FilePos` returns the current file position. Parameter *f* represents an open structured or untyped file.

Returns The current file position.

See also `PLFilePos`

FileSize

Description Returns the size of a file.

Compatibility Turbo Pascal

Prototype

```
FUNCTION FileSize(
  VAR f : FileType) : LONGINT;
```

Remarks `FileSize` returns the size of the file referred to by *f*. Parameter *f* represents an open structured or untyped file.

Unlike the Turbo Pascal version of `FileSize`, the CodeWarrior Pascal version returns the file position if the file position is past the end-of-file. The Turbo Pascal version of `FileSize` will return the actual file size, even if the file position is past the end-of-file.

Returns The number of logical records in file *f*.

See also `FilePos`, `Seek`

Flush

Description Sends unwritten buffer data to a text file.

Compatibility Turbo Pascal

Prototype

```
PROCEDURE Flush(
  VAR f : TEXT);
```

Remarks `Flush` sends any unwritten data to a file. Parameter *f* represents an open text file in write mode.

See also `PLFlush`

Get

Description Reads the next logical record.

Compatibility ANS Pascal, MPW Pascal

Prototype

```
PROCEDURE Get(VAR f : FileVariableType)
```

Remarks

The `Get` procedure does two things: it advances the current file position to the next component, then it reads the new current logical record into the file window variable.

The parameter *f* is a variable reference that refers to a file variable. The file must be open; if it is not, an error occurs.

If a `Get` procedure is performed when no next logical record exists, `Eof(`*f*`)` becomes `true` and the value of *f*`^` becomes undefined.

The `Get` procedure can be used for character-at-a-time I/O. `Get` with a text file differs from `Get` with a structured file only in that a character is not read until a program reads the file window variable. This behavior of `Get` makes it possible to interact with information entered from the keyboard.

NOTE: Do not mix `Get` and `Put` with `Read` and `Write`. Currently they are not compatible.

See also `ByteRead`, `Put`

Listing 7.7 Get, Put, Open, and Seek example

```
PROGRAM Get_Example(input, output);

(* For the ANS Pascal or the MPW Pascal I/O models. *)

VAR
  numberFile : FILE OF INTEGER;
  i : INTEGER;

BEGIN
  Open(numberFile, 'list');
  FOR i := 0 TO 9 DO
  BEGIN
    numberFile^ := i;
    Put(numberFile);
```

```
  END; { FOR }
  Close(numberFile);

  Open(numberFile, 'list');
  FOR i := 9 DOWNTO 0 DO
  BEGIN
    Seek(numberFile, i);
    IF IOResult <> 0 THEN
    BEGIN
      Close(numberFile);
      Halt;
    END; { IF }
    Get(numberFile);
    Write(numberFile^:3);
  END; { FOR }
  Close(numberFile);
END. { Get_Example }

(* Output:

  9  8  7  6  5  4  3  2  1  0

*)
```

InOutRes

Description Returns the status of the last I/O operation.

Prototype

```
VAR
  InOutRes : INTEGER;
```

Compatibility Turbo Pascal

Remarks This variable behaves identically to `IOResult`. It always contains the status of the last I/O operation executed. A value of zero means no error occurred.

See "IOResult" on page 97 for more information.

See also IOResult

IOResult

Description Returns the status of the last I/O operation.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
FUNCTION IOResult : INTEGER;
```

Remarks `IOResult` can be used with files of all kinds, both typed and untyped.

Because files of type `TEXT` are buffered, `IOResult` does not indicate the result of a write operation until the buffer contents are transferred to the external file. A buffer transfer occurs whenever the buffer is full or when there is a call to `Close`.

Beware of trying to access the value of `IOResult` with an I/O operation such as:

```
Reset(f, 'myfile');
WriteLn('IOResult = ', IOResult);
```

In this example, the value of `IOResult` might be set by the `WriteLn` call, not by the `Reset` call.

NOTE: Some I/O operations may set `IOResult` due to an unseen I/O call made within the CodeWarrior Pascal runtime library.

Turbo Pascal

To intercept I/O errors with `IOResult`, make sure runtime input/output error checking is turned off with the `{$I-}` directive.

The Turbo Pascal I/O model uses a special mechanism to detect input/output errors at runtime. To detect an I/O error, a program must turn off error handling with the `{$I-}` directive, call the I/O function, reactivate runtime error handling with the `{$I+}`, then

check the IOResult routine. If IOResult returns 0, no error occurred and the program may safely continue. If IOResult returns a non-zero value, the program should take steps to recover from the error before continuing with other I/O routines.

See Listing 7.8 for an example.

Listing 7.8 Turbo Pascal I/O Model Example

```
PROGRAM TurboPascal_Example(input, output);

(* For Turbo Pascal. *)

VAR
  f : TEXT;
  c : CHAR;

BEGIN
  {$I-} Reset(f); {$I+} (* Error: f hasn't been assigned yet. *)
  IF IOResult <> 0 THEN
    WriteLn('An I/O error has occurred.');
  ELSE
  BEGIN
    Read(f, c); (* This statement won't be executed. *)
    (* ... *)
  END;
END.

(* Output:
  An I/O error has occurred.);
*)
```

Returns The `IOResult` function returns an `INTEGER` value that indicates the result of the most recently performed I/O operation.

If `IOResult` returns zero, it means that the last I/O operation was successful.

Any non-zero result indicates that the last I/O operation was unsuccessful. These non-zero error codes are listed in `MW_Errno.h`, a C

header file which is in a sub-directory of the source code for the Pascal version of the Metrowerks Standard Library (MSL). Note that values that `IOResult` might return depends on the I/O model in which IOResult is running.

See also `HeapResult`, `InOutRes`, Listing 7.7, "Choosing Which I/O Routines to Use" on page 73

Open

Description Opens an existing file or create and open a new file.

Compatibility ANS Pascal, MPW Pascal

Prototype

```
PROCEDURE Open(
  VAR f        : FileType;
  filename : STRING);
```

Remarks `Open` opens an existing file or creates and opens a new file for random, read/write access, setting the file window variable to the first logical record. An existing file is not truncated.

The parameter *f* is a variable reference that refers to a file variable. The file may not already be open. The parameter *filename* is an expression with a string value, which should be a valid file pathname.

The statement `Open(`*f*`,` *filename*`)` opens an existing external file with the name *filename* and associates the file variable specified by *f* with this external file. If an external file with the name *filename* does not already exist, a new empty file is created. The file is opened for both reading and writing.

After `Open(`*f*`,` *filename*`)` is executed, `EOF(`*f*`)` is `TRUE` if the file is empty; otherwise, `EOF(`*f*`)` is `FALSE`. Also, the current file position is logical record 0, and the file window variable *f*`^` contains the value of that logical record, unless `EOF(`*f*`)` is `true`.

`Open` can be used with files of all kinds, both typed and untyped.

See also `Close`, `Reset`, `Rewrite`, Listing 7.7.

Page

Description Outputs a form feed.

Compatibility ANS Pascal, MPW Pascal

Prototype

```
PROCEDURE Page[(VAR f : TEXT)];
```

Remarks The `Page` procedure sends a form feed character (ASCII 12) to the file of `TEXT` type designated by *f*. If the file parameter is omitted, the character is sent to the predefined file `output`.

See also `WriteLn`

PLCrunch

Description Truncates at the current file position.

Compatibility MPW Pascal

Prototype

```
USES PasLibIntf;
PROCEDURE PLCrunch(
  VAR f  : FileType);
```

Remarks The `PLCrunch` procedure takes an open file as an argument. It truncates the file at the current file position. The parameter *f* is a variable reference that refers to a file variable. The file must be open. `PLCrunch` works on any file type.

See also `Reset`, `Rewrite`, `Truncate`.

PLFilePos

Description Returns the record number of the current file position.

Compatibility MPW Pascal

Prototype

```
USES PasLibIntf;
FUNCTION PLFilePos(
  VAR f : FileType) : LONGINT;
```

Remarks The `PLFilePos` function returns the logical record number of the current file position. With untyped files or files of type `TEXT`, it returns the number of bytes from the beginning of the file to the current file position. The parameter *f* is a variable reference that refers to a file variable. The file must be open. `PLFilePos` works on any file type.

Returns `PLFilePos` returns a value of type `LONGINT` that is the logical record number of the current file position.

See also `FilePos`, `Seek`

PLFlush

Description Flushes an output buffer to a file.

Compatibility MPW Pascal

Prototype

```
USES PasLibIntf;
PROCEDURE PLFlush(
  VAR f : TEXT);
```

Remarks The `PLFlush` procedure causes the contents of the current output buffer associated with the file *f* of type `TEXT` to be written to the file.

See also `PLSetVBuf`, `Write`

PLPurge

Description Deletes a file.

Compatibility MPW Pascal

Prototype

```
USES PasLibIntf;
PROCEDURE PLPurge(
  filename      : STRING);
```

Remarks The `PLPurge` procedure deletes the file named by *filename*, a character string. An error occurs if the file is open.

See also `IOResult`, `Erase`

PLRename

Description Renames a file.

Compatibility MPW Pascal

Prototype

```
USES PasLibIntf;
PROCEDURE PLRename(
  oldname   : STRING,
  newname   : STRING);
```

Remarks The `PLRename` procedure allows you to rename any file.

The *oldname* and *newname* parameters are strings. The *oldname* parameter is the pathname of an existing file, which may be open; *newname* is a new pathname. If a file named *newname* already exists, an error occurs and the value of `IOResult` is set to non-zero.

See also `IOResult`

PLSetVBuf

Description Specifies file buffering management.

Compatibility MPW Pascal

Prototype

```
USES PasLibIntf;
PROCEDURE PLSetVBuf(
  VAR f         : TEXT,
```

```
bufptr  : CHARArrayType,
style   : IntegerType,
bufsize : IntegerType);
```

Remarks The `PLSetVBuf` procedure allows you to specify your own buffer for use with files of type `TEXT`.

The parameter *f* is a variable reference that refers to a file variable of type `TEXT`. The parameter *bufptr* is a pointer to a `PACKED ARRAY OF CHAR` to be used as a text I/O buffer. The *bufsize* parameter is an `INTEGER` that gives the size of the buffer in bytes. The *style* parameter is an `INTEGER` that determines how buffering is performed (Table 7.2)

Table 7.2 PLSetVBuf settings

Style ID	Effect
_IOFBF	I/O is file buffered.
_IOLBF	Output is line buffered; the buffer is flushed when full or at `Eoln`.
_IONBF	I/O is unbuffered; *bufptr* and *bufsize* are ignored; file cannot be used.

The system normally allocates a file's buffer when the first read or write operation is performed on it. To allocate your own buffer, call `PLSetVBuf` after the file is opened but before the first read or write operation. If the value of *bufptr* is `NIL`, the system allocates a buffer of size *bufsize* at the first read or write operation. Be sure to close the file before deallocating its buffer.

See also `PLFlush`, `Write`

Put

Description Writes a file variable at the current file position.

Compatibility ANS Pascal, MPW Pascal

Prototype

```
PROCEDURE Put(VAR f : FileVariableType);
```

Remarks

The `Put` procedure does two things. First, it writes the file window variable into the file at the current file position. Second, it advances the file position to the next logical record.

The parameter *f* is a variable reference that refers to a file variable. The file must be open, and the value of *f*^ must not be undefined.

The statement `Put(`*f*`)` writes the value of *f*^ to the external file at the current file position and advances the current file position to the next logical record. If `EOF(`*f*`)` is `TRUE`, `Put(`*f*`)` appends the value of *f*^ to the end of the file *f* and `EOF(`*f*`)` remains `TRUE`.

The `Put` procedure can also be used for character-at-a-time I/O.

NOTE: Don't mix `Get` and `Put` with `Read` and `Write`. Currently they are not compatible.

See also `ByteRead`, Listing 7.7.

Read

Description Reads text or a logical record from a file.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
PROCEDURE Read(
  [VAR f : TEXTOrStructuredFileType;]
  VAR v1 [;
  VAR v2, ...;
  VAR vn]);
```

Remarks

The `Read` procedure reads one or more values from a text file or structured file into one or more variables.

If *f* is given, it must be a variable reference that refers to a file variable of type `TEXT` or a structured file type. The file must be open. If *f* is omitted, it is assumed to be the standard input file.

Table 7.3 Ways to input data

For information on reading...	Refer to this section
From a console and files of type TEXT	"Reading From Standard Input or a File of Type TEXT" on page 105
From a structured file	"Reading Structured Files" on page 109
Boolean values	"Reading Boolean Values" on page 106
Characters	"Reading Characters" on page 106
Integer values	"Reading Integer Values" on page 106
Real values	"Reading Floating Point Values" on page 107
Packed arrays of characters	"Reading PACKED Array of Characters" on page 109
Character strings	"Reading Character Strings" on page 108

Reading From Standard Input or a File of Type TEXT

Each *vn* is a variable reference that refers to a variable of one of the following types:

- `CHAR` or a subrange of type `CHAR`.
- `INTEGER`, `LONGINT`, or a subrange of `INTEGER` or `LONGINT`.
- One of the real types (`REAL`, `LONGREAL`, `SINGLE`, `DOUBLE`).
- A scalar type (not including `BOOLEAN`) or a subrange of a scalar type.
- A `STRING` type.

- A `PACKED ARRAY OF CHAR`.
- A type that is assignment-compatible with the logical records of *f* if *f* is a structured file.

`Read` with an array element follows the rules for the element's type. The other possibilities are discussed later.

Reading Boolean Values

With the MPW Pascal, boolean values may be read from a file. Boolean values may not be directly read with the ANS Pascal or the Turbo Pascal.

If the next few characters to be read are "T" or "True", a boolean value of TRUE will be read. Otherwise, FALSE will be read.

Leading spaces, tabs, and end-of-lines are skipped.

Reading Characters

A `Read` performed with a `CHAR` type variable is considered equivalent to this compound statement:

```
BEGIN
   v := ff^;
   Get(ff)
END
```

In this example, `v` is a variable of type `CHAR` and `ff` is a `FILE OF CHAR`. Remember that if the current file position is at an end-of-line character, `ff^` contains a space character.

Turbo Pascal

If `EOF(ff)` returns true, then reading a character from file `ff` gives the Control-Z character (ASCII 26).

Reading Integer Values

A `Read` procedure performed with an integer data type reads a sequence of characters that form a signed, whole number. If the sequence of characters is a valid representation of the integer type

being read, the value is assigned to the variable. Otherwise, an error occurs.

After a `Read` from a text file with an integer variable, the current file position is the character following the last character in the numeric string, unless the last character in the string was the last character in the file. Also, `EOF(`*f*`)` will return `TRUE` if the last character in the numeric string was the last character on the line or in the file.

ANS Pascal **and** MPW Pascal

When an integer value is being read, the leading spaces, tabs, and end-of-line sequences preceding the first digit or the sign are skipped. Reading ceases as soon as a character is reached that (together with the characters already read) does not form part of a signed whole number; or when `EOF(`*f*`)` becomes `TRUE`.

If a signed whole number is not found after skipping any preceding spaces, tabs, and end-of-line characters, a value of 0 is read and an error occurs.

Turbo Pascal

When an integer value is being read, the leading spaces, tabs, and end-of-line sequences preceding the first digit or sign are skipped. Reading stops when a space, tab, or end-of-line sequence is reached. If the characters read don't form a valid integer value, an I/O error occurs. If no number was reached before reaching the end-of-file, a zero is read.

Reading Floating Point Values

A `Read` performed on a text file with a variable of one of the real types reads a sequence of characters that forms a signed number. If the sequence of characters is a valid representation of a value of a `REAL` type, the value is assigned to the variable. Otherwise, an error occurs.

Immediately after a `Read` from a text file with a variable of the floating point type the current file position is the character following the last character in the numeric string, unless the last character in the string was the last character in the file. Also, `EOF(`*f*`)` will return

`TRUE` if the last character in the numeric string was the last character on the line or in the file.

ANS Pascal and MPW Pascal

When a floating point value is being read, leading blanks, tabs, and end-of-line characters preceding the first digit or the sign are skipped. Reading ceases as soon as a character is reached that (together with the characters already read) does not form part of a signed number, or when `EOF(`*f*`)` becomes `TRUE`.

If a signed floating point number is not found after skipping any preceding spaces and tabs, `NAN` is returned and an I/O error occurs.

Turbo Pascal

When a floating point value is being read, the leading spaces, tabs, and end-of-line sequences preceding the first digit or sign are skipped. Reading ceases stops when a space, tab, or end-of-line sequence is reached. If the characters read don't form a valid floating point value, an I/O error occurs. If no number was reached before reaching the end-of-file, a zero is read.

Reading Character Strings

A `Read` performed with a string variable reads a sequence of characters up to, but not including, the next end-of-line character, or until the end of the file. The resulting character string is assigned to the variable. It is an error if the number of characters read exceeds the size attribute of the variable.

NOTE: With a string variable, `Read` does not skip to the next line after reading; an end-of-line character is left in the file buffer. For this reason, you cannot use successive `Read` calls to read a sequence of strings; after the first `Read`, each subsequent `Read` will access the end-of-line (instead of a character) and will read a zero-length string. Instead, you must use `ReadLn` to read string values; `ReadLn` skips to the beginning of the next line after each input.

After a `Read` from a text file with a string variable, the current file position is the character following the last character in the string, unless the last character in the string was the last character in the file. Also, `EOF(`*f*`)` will return `true` if the last character in the string was the last character in the file. `EOLn(`*f*`)` will return `TRUE` unless `EOF(`*f*`)` is `TRUE`, in which case `EOLn(`*f*`)` is undefined.

Reading PACKED Array of Characters

ANS Pascal **and** MPW Pascal

When reading a `PACKED` array of characters, the number of characters read is the number of elements in the array unless end-of-line or end-of file is reached. Unread elements in the array are assigned space characters (ASCII 32).

Turbo Pascal

When reading a PACKED array of characters, the array's first element must be numbered 0.

The number of characters read is the number of elements in the array unless end-of-file is reached. The element after the last element read is given a NULL value (ASCII 0).

Reading Structured Files

The `Read` procedure is usually used with files of type `TEXT`. When used with a structured file, it reads one or more logical records into one or more variables, starting at the current file position and advancing the current position pointer.

Each parameter v_n must be a variable of a type that is assignment-compatible with the logical records of *f*.

For example, if `NewVals` is a file of type `FILE OF REAL`, then the procedure call

```
Read(NewVals, Subtotal, Total);
```

requires that `Subtotal` and `Total` be variables of types to which a value of the type `REAL` can be assigned. If `NewVals` is a variable of type `INTEGER`, for instance, a compiler error occurs. For the rules

that determine possible types in this context, see the Pascal Language Reference.

See also IOResult, Get, ReadLn, Write, ReadString

Listing 7.9 Read and ReadLn example

```
PROGRAM Read_Example(input, output);

(* For the ANS Pascal or the MPW Pascal I/O models. *)

VAR
  i : INTEGER;
  x : LONGREAL;
  s : STRING[20];
  a : PACKED ARRAY [1 .. 20] OF CHAR;

BEGIN
  FOR i := 1 TO 20 DO
    a[i] := ' '; { Fill array with blanks }

  Write('Enter an integer: ');
  Read(i);
  WriteLn(i:20);
  Write('Enter a real number: ');
  Read(x);
  WriteLn(x:10:10);
  ReadLn;
  Write('Enter a string: ');
  ReadLn(s);
  WriteLn(s:20);
  Write('Enter a string: ');
  ReadLn(a);
  FOR i := 1 TO 20 DO
    Write(a[i]);
  WriteLn;
END. { Read_Example }

(* Output:
```

```
Enter an integer: 100
                  100
Enter a real number: 3.128
3.1280000000
Enter a string: Isabelle now
        Isabelle now
Enter a string: Isabelle then

Isabelle then

*)
```

ReadLn

Description Reads until end-of-line is reached.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
PROCEDURE ReadLn[(
  [VAR f : TEXT;]
  [VAR v1 ;
  VAR v2, ...; VAR vn])];
```

Remarks The `ReadLn` procedure is an extension of `Read`. It reads a sequence of characters until the next character is the end-of-line character. It then skips to the beginning of the next line in the input file. Because `ReadLn` depends on finding the end-of-line character, it can be used only with files of type `TEXT`.

The parameters allowed with `ReadLn` are the same as those for `Read`. In addition, you can use `Readln` with no input variables and/or no parameters

If the first parameter does not specify a file or if no parameters are used, `Readln` reads from the standard file `input`.

When `ReadLn` is used without input variables, it advances the current file position to the beginning of the next line, if there is one. If

there is no next line, it advances the current file position to the end of the file.

After a `ReadLn`, regardless of the type of any input variable used, `EOF(`*f*`)` will return `TRUE` if the line read was the last line in the file and `EOLn(`*f*`)` will return `FALSE` unless the line following the line read is empty.

See also `Read`, `WriteLn`, Listing 7.9.

Rename

Description Changes the name of a file.

Compatibility Turbo Pascal

Prototype

```
FUNCTION Rename(
  VAR f       : FileVariableType,
  newName : STRING);
```

Remarks `Rename` changes the name of the file referred to by *f* with the name in *newName*.

Parameter *f* refers to any kind of file.

See also `PLRename`

Reset

Description Opens an existing file.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype ANS Pascal, MPW Pascal

```
PROCEDURE Reset(
  VAR f        : FileVariableType [;
  filename : STRING]);
```

Turbo Pascal

```
PROCEDURE Reset(
  f         : FileVariableType [;
  blockSize : INTEGER]);
```

Remarks `Reset` can be used with files of all kinds, both typed and untyped.

ANS Pascal

The `Reset` procedure opens an existing file for sequential read-only access or "rewinds" an open file so that its window variable contains the first logical record.

The parameter *f* is a variable reference that refers to a file variable. The parameter *filename* is an optional expression with a string value. If *filename* is given, the file should not already be open.

If *filename* is not given and the file, *f*, is not open, a temporary file is created and opened.

If *filename* is not given and the file is already open, the file is "rewound." The file must have been originally opened with `Open` or `Reset`. If the file was opened with `Open`, it now becomes read-only.

The value of *filename*, if used, must be a valid file pathname.

Notice that `Reset` preserves the contents of an existing file, unlike `Rewrite`, which erases the current contents of any file on which it is used.

An error occurs and `IOResult` returns a non-zero value if there is no existing external file with the name specified by *filename*.

After `Reset(`*f* `[,` *filename*`])` is executed `EOF(`*f*`)` is `TRUE` if the file is empty. Otherwise, `EOF(`*f*`)` is `FALSE`. Also, the current file position is the first logical record of the file (logical record number 0), and the file window variable *f*`^` contains the value of that logical record unless `EOF(`*f*`)` is `TRUE`, in which case the value of *f*`^` is undefined.

MPW Pascal

The `Reset` procedure opens an existing file for sequential read-only access or "rewinds" an open file so that its window variable contains the first logical record.

The parameter *f* is a variable reference that refers to a file variable. The parameter *filename* is an optional expression with a string value. If *filename* is given, the file should not already be open. If *filename* is not given, the file must be open.

The value of *filename*, if used, must be a valid file pathname.

The statement `Reset(`*f*`)`, when the file specified by *f* is already open, causes the file to be "rewound." The file must have been originally opened with `Open` or `Reset`. If the file was opened with `Open`, it now becomes read-only.

Notice that `Reset` preserves the contents of an existing file, unlike `Rewrite`, which erases the current contents of any file on which it is used.

An error occurs and `IOResult` returns a non-zero value if there is no existing external file with the name specified by *filename*.

After `Reset(`*f* `[,` *filename*`])` is executed `EOF(`*f*`)` is `TRUE` if the file is empty. Otherwise, `EOF(`*f*`)` is `FALSE`. Also, the current file position is the first logical record of the file (logical record number 0), and the file window variable *f*`^` contains the value of that logical record unless `EOF(`*f*`)` is `TRUE`, in which case the value of *f*`^` is undefined.

Turbo Pascal

Reset opens an existing file. Parameter *f* is a text, structured, or untyped file. The type of access to the file is specified with the previously-set value of `FileMode`. By default, file access is read/write.

The optional *blockSize* parameter can be used only with untyped files. This parameter specifies the record size to use for reading and writing with `BlockRead` or `BlockWrite`. The default value is 128 bytes.

See also PLCrunch, BlockRead, BlockWrite, Close, FileMode, IOResult, Open, Rewrite, Listing 7.4.

Rewrite

Description Creates and opens a new, empty file.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype ANS Pascal, MPW Pascal

```
PROCEDURE Rewrite(
  VAR f        : FileVariableType [;
  filename : STRING]);
```

Turbo Pascal

```
PROCEDURE Rewrite(
  f          : FileVariableType [;
  blockSize : INTEGER]);
```

Remarks `Rewrite` can be used with files of text, structured, and untyped.

ANS Pascal

The `Rewrite` procedure creates and opens a new, empty file for write-only access or "rewinds" and erases an open file.

The parameter *f* is a file variable. If *filename* is given, the file cannot already be open; if it is, an error occurs. The parameter *filename* is an optional expression with a string value. The string should be a valid file pathname.

If *filename* is not given and the file, *f*, is closed, a temporary file is created and opened for write access.

If *filename* is not given and the file is already open, the file is "rewound" and truncated. The file must have been originally opened with `Open` or `Rewrite`. If the file was opened with `Open`, it now becomes write-only.

`Rewrite(`*f*`,` *filename*`)` creates a new external file with the name *filename* and associates the file variable *f* with this external file. If an external file with the name *filename* already exists, it is deleted and a new, empty file with the same name is created in its place.

After `Rewrite(`*f* `[,` *filename*`])` is executed `EOF(`*f* `)` is `TRUE`, either because the file is new or because the contents have just been erased. Also, the current file position is logical record 0; that is, the first logical record written to the file will become the first logical record of the file. The value of *f* ^ is undefined, and remains undefined until something is written to the file.

MPW Pascal

The `Rewrite` procedure creates and opens a new, empty file for write-only access or "rewinds" and erases an open file.

The parameter *f* is a file variable. If *filename* is given, the file cannot already be open; if it is, an error occurs. The parameter *filename* is an optional expression with a string value. The string should be a valid file pathname.

`Rewrite(`*f* `)`, when the file specified by *f* is already open causes the file to be "rewound"; that is, the current file position is reset to the beginning of the file, and any prior contents of the file are deleted. The file must have been opened with `Open` or `Rewrite`. If the file was originally opened with `Open`, it now becomes write-only. `Reset` followed by `Rewrite` causes an error.

`Rewrite(`*f*`,` *filename*`)` creates a new external file with the name *filename* and associates the file variable *f* with this external file. If an external file with the name *filename* already exists, it is deleted and a new, empty file with the same name is created in its place.

After `Rewrite(`*f* `[,` *filename*`])` is executed `EOF(`*f* `)` is `TRUE`, either because the file is new or because the contents have just been erased. Also, the current file position is logical record 0; that is, the first logical record written to the file will become the first logical record of the file. The value of *f* ^ is undefined, and remains undefined until something is written to the file.

Turbo Pascal

The `Rewrite` procedure creates and opens a new file. If the file referred to by parameter *f* already exists, it is deleted then re-created.

If *f* is a text file, then the file is opened in write-only mode. If *f* is a structured or untyped file, then it is opened in the mode specified by `FileMode`. New files are always created with read and write access.

The optional parameter, *blockSize*, can be used to specify the size of blocks used to transfer data. This parameter is only available for structured and untyped files. If *blockSize* is not used, the default block size is 128 bytes.

See also `PLCrunch`, `BlockRead`, `BlockWrite`, `Close`, `FileMode`, `IOResult`, `Open`, `Rewrite`, Listing 7.3

Seek

Description Moves the file position to any logical record.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
PROCEDURE Seek(
  VAR f : FileVariableType;
  n : LONGINT);
```

Remarks The `Seek` procedure allows you to access any logical record in a file. For ANS Pascal and MPW Pascal, Seek can be used for structured files, text files, and untyped files. For Turbo Pascal, `Seek` can be used with untyped files only.

The parameter *f* is a file variable.

The parameter *n* is an expression with a `LONGINT` value that specifies a logical record number in the file. Logical records are numbered from zero. With untyped files or files of the type `TEXT`, *n* is the byte position from the beginning of the file.

For example,

```
Seek(Names, 18);
```

causes the file window variable associated with file `Names` to point to the nineteenth (19th) record of the file.

ANS Pascal, MPW Pascal

`Seek` sets the current file position to logical record *n* then it reads the new current logical record into the file window variable.

The value of *f*`^` becomes the value of the logical record unless *n* is greater than the number of the last logical record of the file, in which case `EOF(`*f*`)` becomes `TRUE`, `IOResult` is set to a non-zero value, and the value of *f*`^` becomes undefined. Thus, `Seek(f , MaxLONGINT)` always sets the current file position to the end-of-file. `eofErr` is not a fatal error, so you can use it to verify that the current file position has actually been moved to the end of the file.

Turbo Pascal

`Seek` sets the current file position to logical record n. To read the record, use the `Read` routine.

To move the file position to the end of a file represented by f use:

```
Seek(f, FileSize(f));
```

No error occurs if a seek goes beyond the end-of-file.

See also `FileSize`, `FilePos`, `PLFilePos`, `Read`, `ReadLn`, Listing 7.7.

SeekEOF

Description Skips white space (spaces, tabs, and end-of-line characters) to find the end-of-file.

Compatibility Turbo Pascal

Prototype

```
FUNCTION SeekEOF[(
  VAR f : TEXT)] : BOOLEAN;
```

Remarks `SeekEOF` skips spaces, tabs, and end-of-line sequences at the current file position before checking for end-of-file status.

Optional parameter *f* refers to a text file. If *f* is not specified, standard input is used. The file referred to by *f* must be open.

Returns If, after skipping a sequence of spaces, tabs, and end-of-line characters, the end-of-file is reached, `SeekEOF` returns `TRUE`. Otherwise, `SeekEOF` returns `FALSE`.

See also `EOF`, `SeekEOF`

SeekEOLn

Description Skips white space (spaces and tabs) to find the end-of-line.

Compatibility Turbo Pascal

Prototype

```
FUNCTION SeekEOLn[(
  VAR f : TEXT)] : BOOLEAN;
```

Remarks `SeekEOLn` skips spaces and tabs before checking for end-of-line status.

Optional parameter *f* refers to a text file. If *f* is not specified, standard input is used. The file referred to by *f* must be open.

Returns If, after skipping a sequence of spaces and tabs, the end-of-line or end-of-file is reached, `SeekEOLn` returns `TRUE`. Otherwise, `SeekEOLn` returns `FALSE`.

See also `EOLn`, `SeekEOF`

SetTextBuf

Description Assigns a buffer to a file.

Compatibility Turbo Pascal

Prototype

```
FUNCTION SetTextBuf(
  VAR f      : TEXT;
  buffer     : RefType [;
  size       : IntegerType ]);
```

Remarks `SetTextBuf` assigns a buffer, *buffer*, to a text file, *f*.

Parameter *f* must refer to an existing closed text file.

Parameter *buffer* refers to a new file buffer. To prevent *buffer* from being deallocated while the file is being operated on, make sure *buffer* is local to a scope that won't be exited while the file is open.

Parameter *size* is optional. It specifies the size of the buffer referred to by *buffer*. If *size* is not specified, then

```
SizeOf(buffer)
```

is used.

See also PLSetVBuf

Truncate

Description Removes data after the current file position.

Compatibility Turbo Pascal

Prototype

```
FUNCTION Truncate(
  VAR f : TEXT);
```

Remarks `Truncate` removes data after the current file position in file, *f*.

Parameter *f* must refer to an open structured or untyped file.

See also PLCrunch

Write

Description Writes text or a logical record to a file.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
PROCEDURE Write(
  [ VAR f : TEXTOrStructuredFileType;]
  p1 [;
  p2; ...;
  pn]);
```

Remarks The `Write` procedure writes one or more values to a file of type `TEXT` or a structured file. The parameter *f* (if given) is a variable reference that refers to a file variable of type `TEXT` or a structured file type. The file must be open. If *f* is omitted, the procedure writes to the predefined file `output`.

Table 7.4 Finding ways to output data

For information on writing...	Refer to this section
To a console and files of type TEXT	"Writing to Standard Output and Files of Type TEXT" on page 122
To a structured file	"Writing to Structured File Types" on page 126
`BOOLEAN` data types	"Writing BOOLEAN Values" on page 125
Characters	"Writing Characters" on page 122
Integer values	"Writing Integers" on page 123
Real values	"Writing Fixed Point Numbers" on page 124 and "Writing Floating Point Numbers" on page 123
Character strings	"Writing Character Strings" on page 125 and "Writing PACKED ARRAY OF CHAR" on page 125

Writing to Standard Output and Files of Type TEXT

Each p_n parameter is a `Write` parameter. At least one `Write` parameter must be present. The value of each `Write` parameter, p_n, is given by an *output expression*, which may be of type `CHAR`, `INTEGER`, `REAL`, `STRING`, `PACKED ARRAY OF CHAR`, or `BOOLEAN`.

Write is also used to output a type that is assignment-compatible with the logical records of *f* if *f* is a structured file. Using `Write` for output to a structured file is described later.

Each `Write` parameter has the form

```
OutExpr[:MinWidth[:DecPlaces]]
```

where *OutExpr* is an output expression. *MinWidth* and *DecPlaces* are optional expressions with integer values.

MinWidth specifies the minimum field width. *MinWidth* must be greater than or equal to zero. Exactly *MinWidth* characters are written (using leading spaces if necessary), except when *OutExpr* has a value that must be represented in more than *MinWidth* characters, in which case the exact number of characters needed is written. *MinWidth* can be used with an *OutExp* of any type that is valid in a `Write` parameter.

DecPlaces specifies the number of decimal places to be used in the fixed point representation of a floating point value. It can be specified only if *OutExpr* has a floating point type value and if *MinWidth* is also specified. If specified, it must be greater than zero. If *DecPlaces* is not specified and the value is one of the floating point types, a floating point representation is written. Floating point representation is discussed later.

Writing Characters

If `Write` is given a variable of type `CHAR` and *MinWidth* is not specified, the character value of *OutExpr* is written to the specified file. If *MinWidth* is included, exactly *MinWidth*–1 spaces are written, followed by the character value of *OutExpr*.

Writing Integers

If *OutExpr* is of an integer type, its decimal representation is written.

If *MinWidth* is used and its value is greater than the number of digits in the decimal representation of the value to be written, leading spaces will be written to the left of the number. The number of spaces depends upon the *MinWidth* specification.

If the value of *OutExpr* is less than zero, a minus sign (ASCII 45), denoting a negative value, is written to the file and counts as a leading space.

ANS Pascal **and** MPW Pascal

If *MinWidth* is omitted, it is given a default value of eight.

Turbo Pascal

If *MinWidth* is omitted, it is given a default value of zero.

Writing Floating Point Numbers

If *DecPlaces* is absent, a floating point representation is written. To output a floating point representation, *OutExpr*'s value is expressed such that *OutExpr* = m.n*10^{e}.

The corresponding character string format for a floating point number is:

```
{leading spaces}{- | space} m.nE±e
```

If the character string representation of *OutExpr* does not exceed *MinWidth*, leading spaces are prefixed to the output. If the *OutExpr* value is less than zero, a negative sign (**–**) is output, or, if *MinWidth* is not exceeded and *OutExpr* is a positive value, a space is output. If a negative sign (**–**) is output, it is counted as leading space.

ANS Pascal

The *m* expression is a single digit from 0 to 9. The *n* expression is the decimal digits with zeroes padded on the right, if necessary, to make *Minwidth* - 9 digits. The `E` character is then output, a sign digit is output (either **+** or **–**) and then the exponent value *e* is written,

with enough leading zeroes to make the exponent length equal to two.

MPW Pascal

The *m* expression is a single digit from 0 to 9. The *n* expression is the decimal digits with zeroes padded on the right, if necessary, to make *Minwidth* - 9 digits. The `E` character is then output, a sign digit is output (either + or –) and then the exponent value *e* is written, with enough leading zeroes to make the exponent length equal to four.

Turbo Pascal

The *m* expression is a single digit from 0 to 9. The *n* expression is the decimal digits with zeroes padded on the right, if necessary, to fill in *MinWidth* digits. The `E` character is then output, a sign digit is output (either + or –) and then the exponent value *e* is written, with enough leading zeroes to make the exponent length equal to two.

If the floating point value being output is 0, then only 0 is output.

If it is not specified, the default value for *MinWidth* is 17. If it is less than 8, a value of 8 is used.

Writing Fixed Point Numbers

If *DecPlaces* is present, a fixed point representation is written to the file using this format:

```
{leading spaces} { - | space} i.f
```

If the character string representation does not exceed *MinWidth*, leading spaces are prefixed to the output.

If the *OutExpr* value is less than zero, a negative sign (–) is output, or, if *MinWidth* is not exceeded and *OutExpr* is a positive value, a space is output. If a negative sign (–) is output, it is counted as leading space.

The *i*`.`*f* expression signifies the digits before the decimal point (*i*), the decimal point itself (`.`), and the digits after the decimal point (*f*),

padded with zeroes on the right, if necessary, to make *DecPlaces* decimal digits.

ANS Pascal **and** MPW Pascal

If *MinWidth* is omitted from the `Write` parameter, it is assumed to be ten.

Turbo Pascal

If the fixed point value being output is 0, then only 0 is output.

If *MinWidth* is omitted from the `Write` parameter, it is assumed to be 17. If it is less than 11, a value of 11 is used.

Writing Character Strings

The results of using `Write` with a `STRING` variable depend upon the length attribute of the string that appears as the *OutExpr* and whether or not *MinWidth* is specified.

If *MinWidth* is specified and the length of the string is less than *MinWidth*, then the string is written preceded by a number of spaces equal to *MinWidth* minus the length of the string.

If *MinWidth* is specified and the length of the string is greater than *MinWidth*, then the first *MinWidth* number of characters are written.

If *MinWidth* is specified and the length of the string equals *MinWidth*, or if *MinWidth* is not specified, the entire string value is written on the file.

Writing PACKED ARRAY OF CHAR

If *OutExpr* is a `PACKED ARRAY OF CHAR`, the effect is the same as writing a string whose length is the number of elements in the array.

Writing BOOLEAN Values

ANS Pascal **and** MPW Pascal

If the value of *OutExpr* is type `BOOLEAN`, the string "` TRUE`" (with a leading space) or the string "`FALSE`" is written to the file *f*. The de-

fault value of *MinWidth* is five. If *MinWidth* is greater than five, leading spaces are added; if *MinWidth* is less than five, the character "`T`" or "`F`" is written, padded with spaces as if a value of type `STRING[1]`.

Turbo Pascal

If the value of *OutExpr* is type `BOOLEAN`, the string "`True`" or the string "`False`" is written to the file *f*.

Writing to Structured File Types

The `Write` procedure is usually used with files of type `TEXT`. When used with a structured file, it writes the value of one or more variables into one or more logical records of the file, starting at the current file position and advancing the current position pointer.

Each parameter p_n must be a variable of the type that is assignment-compatible with the logical records of *f*. For the rules that determine possible types in this context, see the Pascal Language Reference.

See also `ByteRead`, `IOResult`, `Read`, `Put`, `WriteLn`, `StringOf`

Listing 7.10 Write and WriteLn example

```
PROGRAM Write_Example(input, output);

(* For ANS Pascal and MPW Pascal *)

BEGIN
  Write(1:5);
  Write(' ');
  Write(2.2345:5:5);
  WriteLn(TRUE, ' ',FALSE);
  WriteLn('Laughing like little lumpfish.');
  WriteLn('Bob':6);
  WriteLn('Teri-Ann':6);
  WriteLn('Andy':6);
  WriteLn('Vada':6);
END. { Write_Example }
```

```
(* Output:

    1 2.23450 TRUE FALSE
Laughing like little lumpfish.
   Bob
Teri-A
  Andy
  Vada

*)
```

WriteLn

Description Writes to a text file and terminates with an end-of-line.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
PROCEDURE WriteLn(
  [ VAR f : TEXT;]
  [ p1 [;
  p2; ...;
  pn]]);
```

Remarks The `WriteLn` procedure is an extension of `Write`. It performs the same actions and then writes an end-of-line character to the output file.

The parameters are the same as those used with `Write`, except the `Write` parameters can be entirely omitted; without them, `WriteLn` writes an end-of-line character to the output.

See also `Write`, Listing 7.10.

8

Console Input & Output on Mac OS

When creating standard Pascal programs that use console input/output, CodeWarrior Pascal programs that run on Mac OS transparently calls SIOUX. SIOUX is a library for Mac OS that provides a text window and menus for simple TTY-type input/output.

SIOUX is not available for operating systems that already have console input/output features, like Windows 95/NT 4.

SIOUX Overview

Sometimes you need to port a program to Mac OS that was originally written for MS-DOS™, UNIX™, or some other operating system that uses a character-based user interface. Also, you might need to write a new program quickly and don't have the time to write a complete Mac OS program that handles windows, menus, and events.

Unlike other operating systems like Windows® or UNIX™, Mac OS doesn't have a "built-in" console input/output features. To help you, CodeWarrior Pascal provides the SIOUX library for Mac OS, which handles all the Mac OS menus, windows, and events so your program doesn't need to. It creates a window that's much like a dumb terminal or TTY. You can write to it and read from it with the standard Pascal input/output routines, such as `Write`, `WriteLn`, `Read`, `ReadLn`. SIOUX also provides a **File** menu that lets you save and print the contents of the window, and an **Edit** menu that lets you cut, copy, and paste the contents in the window.

CodeWarrior Pascal, including the SIOUX library, does not support cursor positioning or character attributes such as reverse-video. Routines such as `GotoXY`, `TextBackground`, `TextColor`, and

`TextMode` are not supported. However, SIOUX does allow the window size and display font and size to be customized. See "Customizing SIOUX" on page 132 for more information.

Figure 13.1 shows a running SIOUX program.

Figure 8.1 A Running SIOUX Program

Your program reads and writes text to a resizable, scrolling window. It saves up to 32K of your program's text.

With the commands from the **Edit** menu, you can cut and copy text from the SIOUX window and paste text from other applications into the SIOUX window. With the commands in the File menu, you can print or save the contents of the SIOUX window.

To stop your program at any time, press Command-Period or Control-C. The SIOUX application keeps running so you can edit or save the window's contents. If you want to exit when your program is done or avoid the dialog asking whether to save the window, see "Changing What Happens on Quit" on page 137.

To quit the SIOUX application at any time, choose **Quit** from the **File** menu. If you haven't saved the contents of the window, the application displays a dialog asking you whether you want to save the contents of the window now.

The status line at the top of the SIOUX window shows information about your program. If you want to remove the status line, see "Showing the Status Line" on page 138.

Creating a Project With SIOUX

To use the SIOUX library, create a project from a project stationery document that creates an ANS Pascal project.

If you want only to write to or read from standard input and output, you don't need to call any special functions or include any special interface files. When your program refers to standard input or output, the SIOUX library initializes itself automatically and creates a SIOUX window for it.

NOTE: In this chapter, standard input and standard output refer to the standard Pascal `input` and `output` files. Remember that functions like `ReadLn` and `WriteLn` use standard input and output by default; these symbols do not appear in the `ReadLn` and `WriteLn` parameter lists.

If you want to add SIOUX to a project you already created, the project must contain certain libraries. Add this library to your 68K project:

- `SIOUX.68K.Lib`

Add this library to your PowerPC project:

- `SIOUX.PPC.Lib`

If you want to use a SIOUX window in a program that has its own event loop, you must modify `SIOUXSettings` and call the function `SIOUXHandleOneEvent`. For more information, see "Using SIOUX Windows in Your Own Application" on page 138.

Customizing SIOUX

This following sections describe how you can customize the SIOUX environment by modifying the structure `SIOUXSettings`. SIOUX uses the data fields of `SIOUXSettings` to determine how to create the SIOUX window and environment.

If you want to customize the SIOUX environment, you must include the SIOUX interface file:

```
USES SIOUX;
```

in your source code and modify `SIOUXSettings` before you use standard input or output. As soon as you write or read to the console, SIOUX creates a window based on the values in `SIOUXSettings` and prevents you from modifying it. For more information, see "Customizing SIOUX" on page 132.

NOTE: To customize SIOUX, you must modify `SIOUXSettings` before you call any function that uses standard input or output. If you modify `SIOUXSettings` afterwards, SIOUX does not change its window.

The first three sections, "Changing the Font and Tabs" on page 135, "Changing the Size and Location" on page 136, and "Showing the Status Line" on page 138, describe how to customize the SIOUX window. "Changing What Happens on Quit" on page 137, describes how to modify how SIOUX acts when you quit it. The last section, """Using SIOUX Windows in Your Own Application" on page 138, describes how you can use a SIOUX window in your own Mac OS program.

SIOUXSettings

Table 8.1 summarizes what's in the `SIOUXSettings` structure.

Table 8.1 The SIOUXSettings structure

This field…	Specifies
`initializeTB : BOOLEAN;`	Whether to initialize the Mac OS Toolbox.
`standalone : BOOLEAN;`	Whether to use your own event loop or SIOUX's.
`setupmenus : BOOLEAN;`	Whether to create File and Edit menus for the application.
`autoquitonclose : BOOLEAN;`	Whether to quit the application automatically when your program is done.
`showstatusline : BOOLEAN;`	Whether to draw the status line in the SIOUX window.
`tabspaces : INTEGER;`	If greater than zero, substitute a tab with that number of spaces. If zero, print the tabs.
`column : INTEGER;`	The number of characters per line that the SIOUX window will contain.
`rows : INTEGER;`	The number of lines of text that the SIOUX window will contain.
`toppixel : INTEGER;`	The location of the top of the SIOUX window.
`leftpixel : INTEGER;`	The location of the left of the SIOUX window
`fontid : INTEGER;`	The font in the SIOUX window.

This field…	Specifies
`fontsize : INTEGER;`	The size of the font in the SIOUX window.
`fontface : INTEGER;`	The style of the font in the SIOUX window.

Listing 13.1 contains a small program that customizes a SIOUX window.

Listing 8.1 Customizing a SIOUX Window

```
PROGRAM Custom_SIOUX;

USES SIOUX;

BEGIN
  (* Don't exit the program after it runs or ask
     whether to save the window *)
  SIOUXSettings.autocloseonquit := FALSE;
  SIOUXSettings.asktosaveonclose := FALSE;

  (* Don't show the status line *)
  SIOUXSettings.showstatusline := FALSE;

  (* Make the window large enough to fit 1 line
     of text that contains 12 characters. *)
  SIOUXSettings.columns := 12;
  SIOUXSettings.rows := 1;

  (* Place the top left corner at 5,40. *)
  SIOUXSettings.toppixel := 40;
  SIOUXSettings.leftpixel := 5;

  (* Set the font to be 48-point, bold,
     italic Times. *)
  SIOUXSettings.fontsize := 48;
  SIOUXSettings.fontface := bold + italic;
  SIOUXSettings.fontid := times;
```

```
  WriteLn('Hello World!');
}
```

Changing the Font and Tabs

This section describes how to change how SIOUX handles tabs with the field `tabspaces` and how to change the font with the fields `fontid`, `fontsize`, and `fontface`.

NOTE: The status line in the SIOUX window writes its messages with the font specified in the fields `fontid`, `fontsize`, and `fontface`. If that font is too large, the status line may be unreadable. You can remove the status line by setting the field `showstatusline` to `FALSE`, as described in "Showing the status line" on page 72.

To change the font in the SIOUX window, set `fontid` to one of these values:

- `courier`
- `geneva`
- `helvetica`
- `monaco`
- `newYork`
- `symbol`
- `times`

By default, `fontid` is `monaco`.

To change the character style for the font, set `fontface` to one of these values:

- `normal`
- `bold`
- `italic`

- `underline`
- `outline`
- `shadow`
- `condense`
- `extend`

To combine styles, add them together. For example, to write text that's bold and italic, set `fontface` to `bold + italic`. By default, `fontface` is `normal`.

To change the size of the font, set `fontsize` to the size. By default, `fontsize` is 9.

The field `tabspaces` controls how SIOUX handles tabs. If `tabspaces` is any number greater than 0, SIOUX prints that number of spaces instead of a tab. If `tabspaces` is 0, it prints a tab. In the SIOUX window, a tab looks like a single space, so if you are printing a table, you should set tabspaces to an appropriate number, such as 4 or 8. By default, `tabspaces` is 4.

The sample below sets the font to 12-point, bold, italic New York and substitutes 4 spaces for every tab:

```
SIOUXSettings.fontsize := 12;
SIOUXSettings.fontface := bold + italic;
SIOUXSettings.fontid := newYork;
SIOUXSettings.tabspaces := 4;
```

Changing the Size and Location

SIOUX lets you change the size and location of the SIOUX window.

To change the size of the window, set `rows` to the number of lines of text in the window and set `columns` to the number of characters in each line. SIOUX checks the font you specified in `fontid`, `fontsize`, and `fontface` and creates a window that will be large enough to contain the number of lines and characters you specified. If the window is too large to fit on your monitor, SIOUX creates a window only as large as the monitor can contain.

For example, the code below creates a window that contains 10 lines with 40 characters per line:

```
SIOUXSettings.rows := 10;
SIOUXSettings.columns := 40;
```

By default, the SIOUX window contains 24 rows with 80 characters per row.

To change the position of the SIOUX window, set `toppixel` and `leftpixel` to the point where you want the top left corner of the SIOUX window to be. By setting `toppixel` to 38 and `leftpixel` to 0, you can place the window as far left as possible and just under the menu bar. Notice that if `toppixel` is less than 38, the SIOUX window is under the menu bar. If `toppixel` and `leftpixel` are both 0, SIOUX doesn't place the window at that point but instead centers it on the monitor.

For example, the code below places the window just under the menu bar and near the left edge of the monitor:

```
SIOUXSettings.toppixel := 40;
SIOUXSettings.leftpixel := 5;
```

Changing What Happens on Quit

The fields `autocloseonquit` and `asktosaveonclose` let you control what SIOUX does when your program is over and SIOUX closes its window.

The field `autocloseonquit` determines what SIOUX does when your program has finished running. If `autocloseonquit` is `TRUE`, SIOUX automatically exits. If `autocloseonquit` is `FALSE`, SIOUX continues to run, and you must choose Quit from the File menu to exit. By default, `autocloseonquit` is `FALSE`.

You can save the contents of the SIOUX window at any time by choosing Save from the File menu.

The field `asktosaveonclose` determines what SIOUX does when it exits. If `asktosaveonclose` is `TRUE`, SIOUX displays a dialog asking whether you want to save the contents of the SIOUX win-

dow. If `asktosaveonclose` is `FALSE`, SIOUX exits without displaying the dialog. By default, asktosaveonclose is `TRUE`.

For example, the code below quits the SIOUX application as soon as your program is done and doesn't ask you to save the output:

```
SIOUXSettings.autocloseonquit := TRUE;
SIOUXSettings.asktosaveonclose := FALSE;
```

Showing the Status Line

The field `showstatusline` lets you control whether the SIOUX window displays a status line, which contains such information as whether the program is running, handling output, or waiting for input. If `showstatusline` is `TRUE`, the status line is displayed. If `showstatusline` is `FALSE`, the status line is not displayed. By default, `showstatusline` is `FALSE`.

Using SIOUX Windows in Your Own Application

This section explains how you can limit how much SIOUX controls your program. But first, you need to understand how SIOUX works with your program. You can consider the SIOUX environment to be an application that calls your main program as just another function. Before SIOUX calls the main program, it performs some initialization to set up the Mac OS Toolbox and its menu. After your program completes, SIOUX cleans up what it created. Even while your program is running, SIOUX sneaks in whenever it performs input or output, acting on any menu you've chosen or command key you've pressed.

However, SIOUX lets you choose how much work it does for you. You can choose to handle your own events, set up your own menus, and initialize the Mac OS Toolbox yourself.

When you want to write an application that handles its own events and uses SIOUX windows for easy input and output, set the field standalone to FALSE before you use standard input or output. SIOUX doesn't use its event loop and sets the field `autocloseonquit` to `TRUE` for you, so the application exits as soon as your pro-

gram is done. In your event loop, you need to call the function `SIOUXHandleOneEvent`, described on page 74.

When you don't want to use SIOUX's menus, set the field setupmenus to FALSE. If standalone is also FALSE, you won't be able to create menus, and your program will have none. If standalone is TRUE, you can create and handle your own menus.

When you want to initialize the Mac OS Toolbox yourself, set the field `initializeTB` to `FALSE`. The field standalone does not affect `initializeTB`.

For example, these lines set up SIOUX for an application that handles its own events, creates its own menus, and initializes the Toolbox:

```
SIOUXSettings.standalone := FALSE;
SIOUXSettings.setupmenus := FALSE;
SIOUXSettings.initializeTB := FALSE;
```

SIOUXHandleOneEvent

Description Handles an event for a SIOUX window.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
FUNCTION SIOUXHandleOneEvent(
  VAR userevent : EventRecord)
    : INTEGER; C; EXTERNAL;
```

Remarks Before you handle an event, call `SIOUXHandleOneEvent` so SIOUX can update its windows when necessary. The argument event should be an event that `WaitNextEvent` or `GetNextEvent` returned. The function returns `TRUE` if it handled the event and `FALSE` if it didn't. If event is a NULL pointer, the function polls the event queue until it receives an event.

Returns If it handles the event, `SIOUXHandleOneEvent` returns `TRUE`. Otherwise, `SIOUXHandleOneEvent` returns `FALSE`.

Listing 8.2 Using SIOUXHandleOneEvent()

```
PROCEDURE MyEventLoop;
VAR
  event : EventRecord;
  cursorRgn : RgnHandle;
  gotEvent,
  SIOUXDidEvent : BOOLEAN;

BEGIN
  cursorRgn := NewRgn();

  REPEAT
    gotEvent := WaitNextEvent(everyEvent, &event,
                        MyGetSleep(), cursorRgn);

    (*
     * Before handling the event on your own,
     * call SIOUXHandleOneEvent() to see whether
     * the event is for SIOUX.
     *)
    if (gotEvent)
      SIOUXDidEvent := SIOUXHandleOneEvent(&event);

    if (!SIOUXDidEvent)
      DoEvent(&event);

  UNTIL gDone;
END; { MyEventLoop }
```

SIOUXSetTitle

Description Changes the title of the SIOUX window.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
PROCEDURE SIOUXSetTitle(
  VAR title: Str255);
    C; EXTERNAL;
```

9

Memory & Object Manipulation

This chapter lists routines that handle heap memory management and the manipulation of objects.

Memory Management Routines Overview

The memory management routines are:

Dispose	Deallocate an object or a piece of previously allocated heap.
FreeMem	Resizes a previously allocated item.
GetMem	Allocates a piece of the heap.
HeapResult	Returns the status of the last memory routine.
Mark	Records an allocation for latest release.
MaxAvail	Returns largest contiuguous piece of memory in the heap.
MemAvail	Returns how much heap is left.
Member	Determines an object's class membership.
New	Allocates an object or a piece of the heap.
PLHeapInit	Initializes the heap in a particular way.
PLSetHeapCheck	Controls heap consistency checking.
PLSetHeapType	Controls how to use the heap.
PLSetMErrProc	Sets the error handler for heap routines.

PLSetNonCont	Controls how the heap is extended.
Release	Deallocates from the previously marked allocation.
SizeOf	Returns the size of a data structure or variable.

These procedures are used to manage the heap, a memory area within the application heap zone. The procedure `New` is used for all allocation of heap space by the program. The `Mark` and `Release` procedures are used together to deallocate all of a marked part of heap space. The `Dispose` procedure is used to deallocate a single identified variable. The `HeapResult` function is used to return the status of the preceding dynamic allocation operation.

The `PLHeapInit` procedure lets you specify the size of the heap you wish to use with your program. The `PLSetNonCont`, `PLSetMErrProc`, and `PLSetHeapType` procedures let you control the characteristics of the heap.

Memory Management Routines

Dispose

Description Deallocates an object or a piece of heap memory.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
PROCEDURE Dispose(
  VAR p : PointerOrObjectType);
```

Remarks The `Dispose` procedure deallocates an identified variable previously allocated with `New`.

The parameter *p* is a variable reference that refers to a variable of any pointer type or object type reference variable. It is a variable parameter.

`Dispose` releases the space allocated to a dynamic variable. It is an error if *p* is undefined or `NIL`. After `Dispose` executes, the value of *p* is undefined. All other references to the identified variable or object that was reached through *p* are also undefined.

An object's `Destructor` method isn't called when an object is deallocated with `Dispose`.

NOTE: You must use the `PLHeapInit` or `PLSetHeapType` procedure to set the allowDispose flag to `true` before you use the `Dispose` procedure. In addition, the allowDispose flag must have been `true` at the time that p was established by a call to `New`.

See also `FreeMem`, `PLHeapInit`, `PLSetHeapType`, `New`

FreeMem

Description Deallocates memory.

Compatibility Turbo Pascal

Prototype

```
PROCEDURE FreeMem(
  VAR p : PointerType;
  s     : UnsignedIntegerType);
```

Remarks The `FreeMem` procedure deallocates an identified variable.

The parameter *p* is a variable reference that refers to a variable of any pointer type. It is a variable parameter. The parameter *s* is an unsigned integer value.

`FreeMem` releases *s* bytes of the space allocated to a dynamic variable. It is an error if *p* is undefined or `NIL`. If *s* is equal or greater than the number of bytes currently allocated for *p* then `FreeMem` sets the value of *p* as undefined. If *s* is less than the number of bytes currently allocated to *p* then the allocation to *p* is resized to *s* bytes.

See also `Dispose`, `New`

GetMem

Description Allocates dynamic memory.

Compatibility Turbo Pascal

Prototype

```
PROCEDURE GetMem(
  VAR p : PointerType;
  s     : UnsignedIntegerType);
```

Remarks The `GetMem` procedure allocates *s* bytes of heap memory and assigns to *p* a pointer to that allocation.

The parameter *p* is a variable reference that refers to a variable of any pointer type. The parameter *p* is a variable parameter; it must be a pointer variable.

If the heap does not contain enough free space to allocate the new variable, *p* is set to `NIL` and a subsequent call to the `HeapResult` function will return a non-zero result.

See also `FreeMem`, `HeapResult`, `New`, `SizeOf`.

HeapResult

Description Returns status of a memory allocation operation.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
FUNCTION HeapResult : INTEGER;
```

Returns The `HeapResult` function returns an integer code that reflects the status of the most recent call on `New`, `Mark`, `Release`, `MemAvail`, or `PLHeapInit`. The codes are given in Table 9.1.

Table 9.1 HeapResult return values

Code	Meaning
0	Successful operation
–1052	Invalid pointer
–1053	Insufficient room in heap

See also PLSetMErrProc, MemAvail.

Listing 9.1 HeadResult example

```
PROGRAM HeapResult_Example;

CONST
  { See Table 9.1 for descriptions of these values }
  HeapOK = 0;
  HeapBadPointer = -1052;
  HeapNoRoom = -1053;

TYPE
  BigArray = ARRAY [1 .. 50000] OF LONGINT;

VAR
  myArrayPtr : ^BigArray;

BEGIN
  New(myArrayPtr);
  CASE HeapResult OF
    HeapOK :
      WriteLn('Array succesfully allocated.');
    HeapBadPointer :
      WriteLn('Array pointer is no good.');
    HeapNoRoom :
      WriteLn('Not enough room in the heap.');
  END; { CASE }
END. { HeapResult_Example }

(* Output:
```

```
Array succesfully allocated.

*)
```

Mark

Description Sets a pointer to a heap area.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
PROCEDURE Mark(p : PointerType);
```

Remarks The `Mark` procedure sets a pointer to a heap area.

The parameter *p* is a variable reference that refers to a variable of any pointer type. It is a variable parameter.

`Mark(`*p*`)` causes the pointer *p* to point to the start of the current free area in the heap. The pointer *p* is also placed on a stack-like list for subsequent use with the `Release` procedure.

See also `New`, `Dispose`, `Release`.

MaxAvail

Description Returns the size of the largest contiguous free block in the heap.

Compatibility Turbo Pascal

Prototype

```
FUNCTION MaxAvail : LONGINT;
```

Returns The `MaxAvail` function returns a `LONGINT` that contains the size of the largest contiguous free block in the current heap. It has no parameters. Note that the result of `MaxAvail` can change over time even if the program does not allocate any heap space, because of other memory management activities.

See also MemAvail, SizeOf

MemAvail

Description Returns the amount of available heap space.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
FUNCTION MemAvail : LONGINT;
```

Returns The `MemAvail` function returns a `LONGINT` that gives the maximum possible amount of available heap space. It has no parameters. Note that the result of `MemAvail` can change over time even if the program does not allocate any heap space, because of other memory management activities.

See also HeapResult, SizeOf

Member

Description Determines an object's membership in a class.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
FUNCTION Member(
  obj : ObjectType;
  ObjectTypeIdentifier) : BOOLEAN;
```

Returns The `Member` function returns true if an instantiated object, *obj*, is of a class type *ObjectTypeIdentifier* or a class derived from *ObjectTypeIdentifier*. `Member` returns false if obj is not of class *ObjectTypeIdentifier*, not of a class derived from *ObjectTypeIdentifier*, or has not been instantiated (by calling `New` with *obj* as its parameter).

See also New

Listing 9.2 Member example

```
PROGRAM Member_Example;

TYPE
  ClassA =
  OBJECT
    (* ... *)
  END;

  ClassB =
  OBJECT (ClassA)
    (* ... *)
  END;

VAR
  a : ClassA;
  b : ClassB;

BEGIN
  New(a);
  New(b);
  IF Member(a, ClassA) THEN
    WriteLn('a is a member of ClassA');
  IF Member(b, ClassA) THEN
    WriteLn('b is a member of ClassA');
  IF Member(a, ClassB) THEN
    WriteLn('a is a member of ClassB');
  IF Member(b, ClassB) THEN
    WriteLn('b is a member of ClassB');
  Dispose(a);
  Dispose(b);
END.

(* Output:

a is a member of ClassA
b is a member of ClassA
b is a member of ClassB
```

```
*)
```

New

Description Allocates a new object or dynamic memory.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
PROCEDURE New(
  VAR p : PointerOrObjectType [;
  t1; ...; tn]);
```

Remarks The `New` procedure allocates a new dynamic variable and sets a pointer variable to point to it.

The parameter *p* is a variable reference that refers to a variable of any pointer type. The parameter *p* is a variable parameter; it can be a pointer variable or object type reference variable of any type.

The optional parameters t_1, ..., t_n are identifiers, used only when allocating a variable of record type with variants.

If *p* is a pointer variable, `New(`*p*`)` allocates a new variable of the base type of *p* and makes *p* point to it. The variable can be referenced as *p*`^`.

If the heap does not contain enough free space to allocate the new variable, *p* is set to `NIL` and a subsequent call to the `HeapResult` function will return a non-zero result.

If the base type of *p* is a record type with variants, `New(`*p*`)` always allocates enough space to allow for the largest variant, even if variants are specified. Although CodeWarrior Pascal accepts the form

```
New(p, t1, ..., tn)
```

it still allocates a variable with space for the largest variant, whether it is specified or not in the call to `New`.

The tag values must be constants; they must be listed contiguously and in the order of their declaration. The tag values are not assigned to the tag fields by this procedure.

An object's `Constructor` method isn't called when an object is allocated with `New`.

See also PLHeapInit, Dispose, HeapResult, Mark, Release, SizeOf.

Listing 9.3 New example

```
PROGRAM New_Example;

TYPE
  TokenType = (Number, CharString, None);
  Token = RECORD
        CASE kind : TokenType OF
          Number : (theValue : INTEGER);
          CharString : (theString : STRING[100]);
      END;
  TokenPtr = ^TokenNode;
  TokenNode = RECORD
          t : Token;
          next : TokenPtr;
        END;

FUNCTION GetValue(VAR inStr : STRING;
                  VAR i : INTEGER) : INTEGER;
VAR
  v : INTEGER;
  done : BOOLEAN;
BEGIN
  v := 0;
  done := FALSE;

  WHILE NOT done DO
  BEGIN
    IF inStr[i] IN ['0' .. '9'] THEN
      v := v * 10 + Ord(inStr[i]) - Ord('0');
    Inc(i);
```

```
    done := NOT (inStr[i] IN ['0' .. '9']);
  END; { WHILE }
  GetValue := v;
END; { GetValue}

FUNCTION GetString(VAR inStr : STRING; VAR i : INTEGER) : STRING;
VAR
  str : STRING[100];
  done : BOOLEAN;
BEGIN
  str := '';
  done := FALSE;

  WHILE NOT done DO
  BEGIN
    IF inStr[i] IN ['a' .. 'z', 'A'..'Z'] THEN
      str := Concat(str, inStr[i]);
    Inc(i);
    done := NOT (inStr[i] IN ['a' .. 'z', 'A'..'Z']);
  END; { WHILE }
  GetString := str;
END; { GetString }

VAR
  tokenList, nextToken, prevToken : TokenPtr;
  inputString : STRING[100];
  i : INTEGER;

BEGIN
  { Initialize }
  tokenList := NIL;

  Write('Enter a string to parse: ');
  ReadLn(inputString);

  { Parse }
  i := 1;
  WHILE i <= Length(inputString) DO
  BEGIN
    CASE inputString[i] OF
```

```
      '0' .. '9' : BEGIN
        New(nextToken, Number);
        nextToken^.t.kind := Number;
        nextToken^.t.theValue :=
          GetValue(inputString, i);
        nextToken^.next := tokenList;
        tokenList := nextToken;
      END;
      'a' .. 'z', 'A' .. 'Z' : BEGIN
        New(nextToken, CharString);
        nextToken^.t.kind := CharString;
        nextToken^.t.theString :=
          GetString(inputString, i);
        nextToken^.next := tokenList;
        tokenList := nextToken;
      END;
      OTHERWISE Inc(i);
    END; { CASE }
  END; { WHILE }

  { Traverse }
  nextToken := tokenList;
  WHILE nextToken <> NIL DO
  BEGIN
    IF nextToken^.t.kind = Number THEN
      WriteLn('Number: ', nextToken^.t.theValue)
    ELSE
      WriteLn('String: ', nextToken^.t.theString);
    nextToken := nextToken^.next;
  END; { WHILE }

  { Delete }
  nextToken := tokenList;
  WHILE nextToken <> NIL DO
  BEGIN
    prevToken := nextToken;
    nextToken := nextToken^.next;
    Dispose(prevToken);
  END; { WHILE }
END. { New_Example }
```

```
(* Output:

Enter a string to parse: I ate 13 apples 2 days ago
String: ago
String: days
Number: 2
String: apples
Number: 13
String: ate
String: I

*)
```

PLHeapInit

Description Initializes a heap.

Compatibility MPW Pascal

Prototype

```
USES PasLibIntf;
PROCEDURE PLHeapInit(
  sizeHeap     : LONGINT;
  heapDelta    : LONGINT;
  memerrProc   : ProceduralPointer;
  allowNonCont : BOOLEAN;
  forDispose   : BOOLEAN);
```

Remarks The `PLHeapInit` procedure initializes the heap, using information you supply to determine its characteristics.

The *sizeHeap* parameter takes an `LONGINT` value that represents the size of the heap. CodeWarrior Pascal's built-in heap-initialization routine automatically allocates 5000 bytes of heap space. Using `PLHeapInit`, you can specify a heap size other than 5000 bytes.

The *heapDelta* parameter specifies the size in bytes of additional space to be added to the heap if *allowNonCont* is `true`. `New` allocations may not be adjacent to the existing heap.

The *memerrProc* parameter is a procedure pointer that enables you to specify a routine to be executed if a memory error, such as heap overflow, occurs.

The *allowNonCont* parameter is a `boolean` value. If it is set to `TRUE`, additional heap space equal to the parameter given to *heapDelta* will be allocated when the initial heap space is exhausted. If it is set to `FALSE`, `PLHeapInit` will ignore *heapDelta*, and no additional space is allocated.

The `forDispose` parameter must be set to `TRUE` if you want to use the `Dispose` procedure to deallocate memory. The default setting is `FALSE`.

`PLHeapInit` should be called by your main program before you allocate any dynamic memory.

See also `Dispose`, `PLSetHeapCheck`, `PLSetHeapType`, `PLSetNonCont`.

PLSetHeapCheck

Description Activates or deactivates heap consistency checking.

Compatibility MPW Pascal

Prototype

```
USES PasLibIntf;
PROCEDURE PLSetHeapCheck(DoIt: BOOLEAN);
```

Remarks Whenever heap space is allocated or deallocated, a consistency check is normally performed on the heap. The procedure `PLSetHeapCheck` allows you to suspend this checking process by setting the `boolean` parameter *DoIt* to `false`. It remains suspended until a subsequent `PLSetHeapCheck` procedure is made with *DoIt* `true`.

`PLSetHeapCheck` should be called by your main program.

See also `PLHeapInit`, `PLSetMErrProc`.

PLSetHeapType

Description Sets the use of `Dispose`.

Compatibility MPW Pascal

Prototype

```
USES PasLibIntf;
PROCEDURE PLSetHeapType(forDispose: BOOLEAN);
```

Remarks The `PLSetHeapType` procedure lets you specify, without accessing `PLHeapInit`, whether or not use of the `Dispose` procedure is to be allowed in your program. If the `BOOLEAN` *forDispose* is `TRUE`, `Dispose` is allowed.

NOTE: Pointers allocated for one type of heap are not compatible with pointers allocated for the other type of heap.

See also `PLHeapInit`, `PLSetNonCont`, `New`

PLSetMErrProc

Description Specifies a memory error handling procedure.

Compatibility MPW Pascal

Prototype

```
USES PasLibIntf;
PROCEDURE PLSetMErrProc(
  errProc : ProceduralPointerType);
```

Remarks The `PLSetMErrProc` procedure allows you to specify a procedure to be executed in case of a memory error. The parameter *errProc* points to the procedure.

See also `HeapResult`.

PLSetNonCont

Description Sets the additional heap space flag.

Compatibility MPW Pascal

Prototype

```
USES PasLibIntf;
PROCEDURE PLSetNonCont(allowNonCont: BOOLEAN);
```

Remarks The `PLSetNonCont` procedure lets you set the additional heap space flag without calling `PLHeapInit`. The *allowNonCont* parameter is a `BOOLEAN` value. If it is set to `true`, additional heap space will be allocated if the current heap is full and cannot be extended.

See also `PLHeapInit`, `PLSetHeapCheck`.

Release

Description Deallocates a marked heap area.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
PROCEDURE Release(p : PointerType);
```

Remarks The `Release` procedure deallocates all variables in a marked heap area.

The parameter *p* is a variable reference that refers to a pointer variable. It must be a pointer that was previously set with the `Mark` procedure.

`Release(`*p*`)` deallocates all areas allocated since the pointer *p* was passed to the `Mark` procedure.

See also `Dispose`, `Mark`, `New`.

SizeOf

Description Returns the size of a variable or type.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
FUNCTION SizeOf(id [;
  t1; ...; tn]) : LONGINT;
```

Returns The `SizeOf` function returns an `LONGINT` value that is the number of bytes occupied by a specified variable, or by any variable of a specified type. Its parameter *id* is either a variable identifier or a type identifier.

The `SizeOf` function always returns the largest size of a variant record, even when the optional parameters t_1, ..., t_n, tag identifiers in a variant record type, are used. These optional parameters may be specified only if the first parameter is a type identifier.

See also `New`, `MemAvail`.

Listing 9.4 SizeOf example

```
PROGRAM SizeOf_Example;

TYPE
  ShapeType = (Sphere, Box);
  IntPtr = ^INTEGER;

VAR
  c : CHAR;
  s : STRING[100];
  a : ARRAY [1 .. 100] OF CHAR;
  pa : PACKED ARRAY [1 .. 100] OF CHAR;
  i : INTEGER;

BEGIN
  WriteLn( SizeOf(ShapeType));
  WriteLn( SizeOf(IntPtr));
  WriteLn( SizeOf(LONGINT));
```

```
  WriteLn( SizeOf(REAL));
  WriteLn( SizeOf(c));
  WriteLn( SizeOf(s));
  WriteLn( SizeOf(a));
  WriteLn( SizeOf(pa));
  WriteLn( SizeOf(i));
END. { SizeOf_Example }

(* Output:

1
4
4
4
2
102
200
100
2

*)
```

10 Ordinal Operations

The ordinal routines operate on the ordinal value of scalar and pointer types, as explained in the Pascal Language Reference.

Ordinal Routines Overview

The ordinal operations are:

Addr	Returns the address of an item.
Chr	Converts a numeric value to a character.
Dec	Decrements a value.
Inc	Increments a value.
Odd	Returns false if a value is even.
Ord	Returns the position of a value within its domain.
Ord4	Converts a pointer to a LONGINT value.
Pointer	Converts a LONGINT value to a pointer.
Pred	Returns a value's predecessor.
Succ	Returns a value's successor.
UpCase	Converts an alphabetic character to upper case.

Ordinal Operation Routines

Addr

Description Returns the address of an item.

Compatibility Turbo Pascal

Prototype

```
FUNCTION Addr(VAR x : DataType) : PointerType;
```

Returns The Addr function returns the address of the parameter passed to it. Its parameter, x, must be a variable parameter.

Remarks The Addr function provides the same feature as the @ operator. In other words, the statement

```
myPtr := @x;
```

is equivalent to

```
myPtr := Addr(x);
```

See also Ord4, Pointer

Listing 10.1 Addr example

```
PROGRAM Addr_example(input, output);

VAR
  a : INTEGER;
  b : ^INTEGER;

BEGIN
  a:= 10;
  WriteLn(a);
  b := Addr(a);
  b^ := 20;
  WriteLn(a);
END.

(* Output:

10
20

*)
```

Chr

Description Returns the CHAR value of a numeric value.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
FUNCTION Chr(x : IntegerType) : CHAR;
```

Returns The Chr function returns the CHAR value corresponding to a whole-number value. Its parameter *x* is an expression with an INTEGER or LONGINT value. For any CHAR value ch, the following is true:

```
Chr(Ord(ch)) = ch
```

Chr(*x*) returns the CHAR value whose ordinal number (that is, its ASCII code) is *x*, if *x* is in the range 0..255. If *x* is not in the range 0..255, the value returned undefined and may result in incorrect program execution.

See also Ord, UpCase, ReadString.

Listing 10.2 Chr example

```
PROGRAM Chr_example;

CONST

VAR
  digit : INTEGER;

BEGIN
  FOR digit := Ord('0') TO Ord('9') DO
    Write(Chr(digit), ' ');
END.

(* Output:

0 1 2 3 4 5 6 7 8 9
```

```
*)
```

Dec

Description Decrements an ordinal variable.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype For ANS Pascal, MPW Pascal:

```
PROCEDURE Dec(VAR x : OrdinalType);
```

For Turbo Pascal:

```
PROCEDURE Dec(
  VAR x : OrdinalType [;
  d     : IntegerType ]);
```

Remarks The `Dec` procedure decrements its variable parameter, *x*, to its previous ordinal value. The *x* parameter must be a variable of ordinal type.

`Dec(`*x*`)` is equivalent to the statement *x* `:= Pred(`*x*`)`.

The other form, `Dec(`*x*`, `*d*`)`, decrements *x* by a value of *d*.

See also Inc, Ord, Pred.

Listing 10.3 Dec and Inc example

```
PROGRAM Dec_example;

VAR
  i : INTEGER;

BEGIN
  i := 1;

  WHILE i <= 5 DO
```

```
  BEGIN
    Write(i);
    Inc(i);
  END;

  WriteLn;
  Dec(i);

  WHILE i >= 1 DO
  BEGIN
    Write(i);
    Dec(i);
  END;

  WriteLn;
  WriteLn;

  i := 8;

  WHILE i >= 0 DO
  BEGIN
    Write(i);
    Dec(i, 2);
  END;

  Inc(i, 2);
  WriteLn;

  WHILE i <= 8 DO
  BEGIN
    Write(i);
    Inc(i, 2);
  END;

END.

(* Output:

       1       2       3       4       5
       5       4       3       2       1
```

```
        8       6       4       2       0
        0       2       4       6       8
*)
```

Inc

Description Increments an ordinal variable.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype For ANS Pascal, MPW Pascal:

```
PROCEDURE Inc(VAR x : OrdinalType);
```

For Turbo Pascal:

```
PROCEDURE Inc(
  VAR x : OrdinalType [;
  i     : IntegerType ]);
```

Remarks The `Inc` procedure increments its variable parameter, `x`, to its next ordinal value. The *x* parameter must be a variable of ordinal type.

`Inc(`*x*`)` is equivalent to the statement `x := Succ(`*x*`)`.

The other form, `Inc(`*x*`, `*d*`)`, increments *x* by a value of *d*.

See also `Dec`, `Ord`, `Succ`, Listing 10.3.

Odd

Description Determines if a whole-number value is odd.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
FUNCTION Odd(x : IntegerType) : BOOLEAN;
```

Remarks The `Odd` function tests whether a whole-number value is odd, returning a `boolean` value. Its parameter *x* is an expression with a value of type `INTEGER` or `LONGINT`.

Returns `Odd(`*x*`)` returns `TRUE` if *x* is odd; if *x* is even, it yields `FALSE`.

See also `Abs`, `Ord`.

Listing 10.4 Odd example

```
PROGRAM Odd_Example;

VAR
  i : INTEGER;

BEGIN
  FOR i := 1 TO 5 DO
    WriteLn(Odd(i));
END. { Odd_Example }

{* Output:

 TRUE
FALSE
 TRUE
FALSE
 TRUE

*)
```

Ord

Description Returns the ordinal number of a scalar or pointer type.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
FUNCTION Ord(
  x : ScalarOrPointerType) : IntegerType;
```

Remarks The `Ord` function returns the ordinal number of a scalar type or pointer type value. Its parameter *x* is an expression with a value of scalar type or pointer type.

For an explanation of the difference between the Ord and Ord4 routines, see the `Ord4` routine description.

Returns If *x* is of type `INTEGER` or `LONGINT`, `Ord` returns the same value as *x*.

If *x* is of pointer type, the result is the corresponding physical address of type `LONGINT`.

If *x* is of another scalar type, the result is the ordinal number determined by mapping the values of the type onto consecutive non-negative whole numbers starting at zero.

For a parameter of type `CHAR`, the result is the corresponding ASCII code.

For a parameter of type `BOOLEAN`, `Ord(FALSE)` returns zero, `Ord(TRUE)` returns one.

See also `Chr`, `Ord4`, `Pred`, `Succ`.

Listing 10.5 Ord, Ord4, and Pointer example

```
PROGRAM Ord_Example;

(* For the MPW Pascal I/O model. *)

TYPE
  Colors = (Red, Blue, Green);

VAR
  myColor : Colors;
  abc, xyz : INTEGER;
  myPtr : ^INTEGER;
```

```
  temp : LONGINT;

BEGIN
  myColor := Blue;
  abc := 1000;
  xyz := 0;
  myPtr := @abc;

  WriteLn( Ord(myColor) );
  WriteLn( Ord4(myColor) );

  WriteLn( Ord(myPtr) );
  WriteLn( Ord4(myPtr) );

  temp := Ord4(@xyz);
  myPtr := Pointer(temp);
  myPtr^ := 1000;

  WriteLn( abc * xyz );
  WriteLn( Ord(abc) * xyz );
  WriteLn( Ord4(abc) * xyz );
END. { Ord_Example }

(* Output:

1
1
14275534
14275534
16960
16960
1000000

*)
```

Ord4

Description Converts a scalar or pointer type to a `LONGINT` type.

Compatibility MPW Pascal

Prototype

```
FUNCTION Ord4(
  x : ScalarOrPointerType) : LONGINT;
```

Remarks The `Ord4` function converts a scalar type or pointer type value to type `LONGINT`. Its parameter *x* is an expression with a value of scalar type or pointer type.

If *x* is of type `LONGINT`, the result is the same as *x*.

If *x* is a pointer type, the result is the corresponding physical address of type `LONGINT`.

If *x* is of type `INTEGER`, the result is the same numerical value represented by *x* but of type `LONGINT`. This is useful in arithmetic expressions. For example, consider the expression

```
abc*xyz
```

where both `abc` and `xyz` are of type `INTEGER`. By the rules given in the Pascal Language Reference, the result of this multiplication is of type `INTEGER` (16 bits). If the mathematical product of `abc` and `xyz` cannot be represented in 16 bits, the result is the low-order 16 bits. To avoid this, the expression can be written as

```
Ord4(abc)*xyz
```

This expression causes 32-bit arithmetic to be used, and the result is a 32-bit `LONGINT` value.

If *x* is of a scalar type other than `INTEGER` or `LONGINT`, the numerical value of the result is the ordinal number determined by mapping the values of the type onto consecutive non-negative integers starting at zero.

Returns `Ord4(`*x*`)` returns the value of *x*, converted to type `LONGINT`.

See also `Ord`, `Pointer`, Listing 10.5.

Pointer

Description Converts an `INTEGER` or `LONGINT` type to a pointer type.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
FUNCTION Pointer(x : IntegerType) : PointerType;
```

Remarks The `Pointer` function converts an `INTEGER` or `LONGINT` value to a pointer type. Its parameter *x* is an expression with a value of type `INTEGER` or `LONGINT`. The pointer returned is of the same type as `NIL` and is assignment-compatible with any pointer type. The value of `Pointer(0)` is `NIL`.

Returns `Pointer(`*x*`)` returns a pointer value that corresponds to the physical address *x*.

See also `Ord4`, `SizeOf`, Listing 10.5.

Pred

Description Returns the predecessor of a scalar value.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
FUNCTION Pred(x : ScalarType) : ScalarType;
```

Returns The `Pred` function returns the predecessor of its parameter *x*, a value of a scalar type.

`Pred(`*x*`)` returns the predecessor of *x* if such a value exists according to the inherent ordering of values in the type of *x*.

If *x* is the first value in the type of *x*, it has no predecessor. In this case, the value returned is not within the range of the type of *x*, and any attempt to assign it to a variable of this type will cause unspecified results.

See also `Dec`, `Succ`.

Listing 10.6 Pred and Succ example

```
PROGRAM Pred_Example;

TYPE
  ScalarType = (Head, One, Two, Three, Four, Five, Tail);

VAR
  a : ScalarType;

BEGIN
  a := Head;
  WHILE Succ(a) <> Tail DO
  BEGIN
    a := Succ(a);
    Write(Ord(a):3);
  END; { WHILE }

  WHILE Pred(a) <> Head DO
  BEGIN
    a := Pred(a);
    Write(Ord(a):3);
  END;
END. { Pred_Example }

(* Output:

  1  2  3  4  5  4  3  2  1

*)
```

Succ

Description Returns the successor of a scalar value.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
FUNCTION Pred(x : ScalarType) : ScalarType;
```

Returns

The `Succ` function returns the successor of its parameter *x*, a value of a scalar type.

`Succ(`*x*`)` returns the successor of *x* if such a value exists according to the inherent ordering of values in the type of *x*.

If *x* is the last value in the type of *x*, it has no successor. In this case, the value returned is not within the range of the type of *x*, and any attempt to assign it to a variable of this type will cause unspecified results.

See also

`Inc`, `Pred`, Listing 10.6.

UpCase

Description

Converts a character to its uppercase equivalent.

Compatibility

ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
PROCEDURE UpCase(x : CHAR) : CHAR;
```

Remarks

The `UpCase` function converts a character, *x*, to its uppercase equivalent.

Returns

`UpCase` returns the uppercase equivalent of *x* if *x* is a lowercase letter.

`UpCase` returns *x* unchanged if *x* is not a lowercase letter.

See also

`Chr`.

Listing 10.7 UpCase example

```
PROGRAM UpCase_Example;

VAR
  s : STRING[100];
```

```
  i : INTEGER;

BEGIN
  Write('Enter a string: ');
  ReadLn(s);
  FOR i := 1 TO Length(s) DO
    s[i] := UpCase(s[i]);
  WriteLn(s);
END. { UpCase_Example }

(* Output:

Enter a string: 2 + 2 equals 4
2 + 2 EQUALS 4

*)
```

11

String Manipulation

String manipulation routines modify character strings and `PACKED` arrays of characters.

String Manipulation Routines Overview

The string procedures and functions now accept `PACKED ARRAY OF CHAR` parameters, but they do not accept indexed string parameters.

The character string manipulation routines and data structure are:

`Concat`	Connects two or more strings together.
`Copy`	Returns part of a string.
`Delete`	Removes part of a string.
`Insert`	Inserts a string into another.
`Length`	Returns the length of a string.
`PLStringOfGet`	Retrieves information on how `StringOf` formats character strings.
`PLStringOfSet`	Specifies how `StringOf` formats character strings.
`Pos`	Finds a string within another.
`ReadString`	Reads input data from a string.
`Str`	Converts a numeric value to a string.
`StringOf`	Converts a string to a value.

StrOf	Describes how StringOf formats character strings.
Val	Converts a string to a numeric value.

String Manipulation Routines

Concat

Description Combines character strings into one character string.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
FUNCTION Concat(
  str1 : StringOrCharacterType;
  str2 : StringOrCharacterType [;
  str3 : StringOrCharacterType; ...]) : STRING;
```

Remarks `Concat` concatenates all the parameters in the order in which they are written. Each parameter is an expression with a value of type `STRING` or `CHAR`. Any number of parameters may be passed. Note that the number of characters in the result cannot exceed 255.

CodeWarrior Pascal also allows the use of the + operator to "add" strings. For example, the statement

```
result := str1 + str2 [+ ... + strn];
```

is equivalent to

```
result := Concat(str1, str2, [..., strn]);
```

where *str1*, *str2*, ..., *strn* are 2 or more character strings.

Returns `Concat` returns the concatenated string.

See also Copy, Delete, Insert.

Listing 11.1 Concat example

```
PROGRAM Concat_example;

CONST
  Greeting = 'Nice to meet you, ';

VAR
  name, result1, result2 : STRING[255];

BEGIN
  Write('Your name, please: ');
  ReadLn(name);

  result1 := Concat(Greeting, name, '!');
  result2 := 'I hope you are well, ' + name + '.';

  WriteLn(result);
END.

(* Output:

Your name, please: Stephen
Nice to meet you, Stephen!
I hope you are well, Stephen.

*)
```

Copy

Description Copies a character string.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
FUNCTION Copy(
  source : STRING;
  index  : IntegerType;
  count  : IntegerType) : STRING
```

Remarks Copy returns a string containing characters from *source*. The parameter *source* is an expression with a value of type `string`. The parameter *index* is an expression with an integer value in the range 1..255. The parameter *count* is an expression with an integer value in the range 1..255.

If the values of *index* or *count* are out of range or if there are not *count* characters in *source* starting at *source* [*index*], Copy returns the null string.

Returns Copy returns a string containing *count* characters from *source*, beginning at *source* [*index*].

See also Concat, Delete, Insert.

Listing 11.2 Copy example

```
PROGRAM Copy_example;


VAR
  name, firstName, lastName : STRING[255];
  endName, lenName : INTEGER;

BEGIN
  Write('Your first and last name: ');
  ReadLn(name);

  { Get length of name }
  lenName := Length(name);

  { Find break between first and last name }
  endName := Pos(' ', name);

  firstName := Copy(name, 1, endName);
  lastName := Copy(name, endName+1,
                   lenName-endName);
  WriteLn('First name: ', firstName);
  WriteLn('Last name: ', lastName);
END.
```

```
(* Output:

Your first and last name: Edbert Cuthbuttle
First name: Edbert
Last name: Cuthbuttle

*)
```

Delete

Description Removes a substring from a character string.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
PROCEDURE Delete(
  VAR dest : STRING;
  index,
  count : IntegerType);
```

Remarks `Delete` removes *count* characters from the value of *dest*, beginning at *dest* [*index*]. The parameter *dest* is a variable reference that refers to a variable of type `string`. It is a variable parameter. The parameter *index* is an expression with an `INTEGER` value in the range 1..255. The parameter *count* is an expression with an integer value in the range 1..255.

If the values of *index* or *count* are out of range or if *index* is greater than `Length(`*dest*`)`, `Delete` is ignored. If the attempted deletion extends beyond the end of *dest*, *dest* becomes truncated at *index* –1.

See also `Concat`, `Copy`, `Insert`, `Pos`.

Listing 11.3 Delete example

```
PROGRAM Copy_example;


VAR
```

```
  name : STRING[255];
  endName : INTEGER;

BEGIN
  Write('Your first and last name: ');
  ReadLn(name);

  { Find break between first and last name }
  endName := Pos(' ', name);

  { Delete the first name }
  Delete(name, 1, endName);

  WriteLn('Last name: ', name);
END.

(* Output:

Your first and last name: Edbert Cuthbuttle
Last name: Cuthbuttle

*)
```

Insert

Description Inserts a character string into a character string.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
PROCEDURE Insert(
  VAR source
  dest        : STRING;
  index       : IntegerType);
```

Remarks `Insert` inserts *source* into *dest*. The parameter *source* is an expression with a value of type `STRING`. The parameter *dest* is a variable reference that refers to a variable of type `STRING`. It is a variable pa-

rameter. The parameter *index* is an expression with an integer value in the range 1..255.

If the value of *index* is out of range, `Insert` does nothing.

See also `Concat`, `Copy`, `Delete`, `Pos`.

Listing 11.4 Insert and Pos example

```
PROGRAM Insert_Example;

VAR
  name,
  sentence : STRING[100];
  p : INTEGER;

BEGIN
  sentence := 'Hello, *. I am pleased to meet you.';
  Write('Enter your name: ');
  ReadLn(name);
  p := Pos('*', sentence);
  Delete(sentence, p, 1);
  Insert(name, sentence, p);
  WriteLn(sentence);
END. { Insert_Example }

(* Output:

Enter your name: Dana
Hello, Dana. I am pleased to meet you.

*)
```

Length

Description Computes the length of a character string array.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
FUNCTION Length(str : STRING) : INTEGER;
```

Returns The `Length` function returns the length of its parameter *str*, which must have a value of type `STRING`.

See also `Pos`, `Concat`, Listing 11.2.

PLStringOfGet

Description Retrieves the settings that the `StringOf` routine uses to format character strings.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
USES PLStringOf;

PROCEDURE PLStringOfGet(
  strofset     : StrOf);
```

Remarks Use the `PLStringOfGet` and `PLStringOfSet` routines and the `StrOf` record type to customize the default settings that `StringOf` uses to format character strings.

When called, `PLStringOfGet` retrieves `StringOf`'s default formatting information and stores it in a variable parameter of type `StrOf`.

After retrieving the formatting settings with `PLStringOfGet`, modify the settings in the `StrOf` record, then set `StringOf`'s default formatting settings with the `PLStringOfSet` routine. Subsequent calls to `StringOf` will use the new settings when formatting items without width or precision specifications.

`PLStringOfGet` is not part of the regular Pascal runtime library. To refer to `PLStringOfGet`, you must add the `PLStringOf` interface to your program's `USES` clause and add the `PLStringOf.p` file to your project.

See also `StringOf`, `PLStringOfSet`, `StrOf`, `Write`.

Listing 11.5 PLStringOfGet, PLStringOfSet, StrOf example

```
PROGRAM StrOf_Example(input, output);

USES PLStringOf; (* Declare PLStringOfGet, PLStringOfSet, StrOf *)

VAR
  aStr : STRING;
  strOfSet : StrOf;

BEGIN

  (* Default behavior for StringOf *)
  aStr := StringOf(1);     (* Assigns '       1' to aStr *)
  aStr := StringOf(1:4);   (* Assigns '   1' to aStr *)

  (* Customizing default behavior for StringOf *)
  PLStringOfGet(strOfSet);
  strOfSet.nbrs.width := 1; (* Change width for integer values *)
  PLStringOfSet(strOfSet);

  aStr := StringOf(1);     (* Assigns '1' to aStr *)
  aStr := StringOf(1:4);   (* Assigns '   1' to aStr *)
END.
```

PLStringOfSet

Description Specifies how the StringOf routine formats character strings.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
USES PLStringOf;

PROCEDURE PLStringOfSet(
  strofset     : StrOf);
```

Remarks The PLStringOfSet routine specifies how subsequent calls to the StringOf routine formats character representations of binary values.

To retrieve the default settings that StringOf uses to format character strings, use the PLStringOfGet routine. After modifying the contents of the StrOf record retrieved by PLStringOfGet, call PLStringOfSet with the modified record to change StringOf's default settings. Subsequent calls to StringOf will use the new settings when formatting items that don't have width or precision specifications.

PLStringOfSet is not part of the Pascal runtime library. To refer to PLStringOfSet, you must add the PLStringOf interface to your program's USES clause and add the PLStringOf.p file to your project.

See also PLStringOfGet, StringOf, Write, Listing 11.5.

Pos

Description Searches for a character string within another character string.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
FUNCTION Pos(
  substr,
  str    : STRING) : INTEGER;
```

Remarks The Pos function searches for *substr* within *str* . Both parameters must be of type STRING.

Returns Pos returns an INTEGER value that is the index of the first character of *substr* within *str*.

If *substr* is not found, Pos returns zero.

See also Concat, Copy, Listing 11.4.

ReadString

Description Reads text from a string.

Compatibility ANS Pascal, MPW Pascal

Prototype

```
PROCEDURE ReadString(
  str : STRING;
  VAR v1; VAR v2; ...; VAR vn);
```

Remarks The `ReadString` procedure takes a character string as input, `str`, and stores the values into its parameters. `ReadString` interprets strings the same way the `Read` routine interprets file or console input.

See also `StringOf`, `Read`.

Listing 11.6 ReadString example

```
PROGRAM ReadString_Example;

VAR
  i : INTEGER;
  x : REAL;
  s : STRING[20];

BEGIN
  ReadString('10 10.123 Michael', i, x, s);
  WriteLn(i, x, s);
END. { ReadString_Example }
```

Str

Description Returns the character string representations of numerical value.

Compatibility Turbo Pascal

Prototype

```
PROCEDURE Str(
  value       : IntegerOrRealType
[: width      : IntegerType
[: precision  : IntegerType ] ] ;
  VAR s       : StringType);
```

Remarks The `Str` function converts a numerical value, *value*, into a character string representation and stores it in *s*.

`Str` accepts parameters like the `Write` procedure, including width and precision specifications.

For example,

```
Str(383, s);
```

stores `'383'` in `s`. In this case, `Str` used its default formatting settings for Turbo Pascal, since width and precision values weren't specified.

The statement

```
Str(3.14:12:4, s)
```

stores `'      3.1400'` in `s` because a width of 12 and a precision of 4 were specified.

For a complete description of how `Str` handles its parameters, see "Write" on page 120.

See also `PLStringOfGet`, `PLStringOfSet`, `StringOf`, `StrOf`, `Write`.

StringOf

Description Returns the character representations of its arguments as a character string.

Compatibility ANS Pascal, MPW Pascal

Prototype

```
FUNCTION StringOf(
  v1 [;
  v2; ...; vn]) : STRING;
```

Remarks The `StringOf` function converts its arguments into character representations and concatenates them, in order, into a item of type `STRING`.

`StringOf` accepts parameters like the `Write` procedure, including width and precision specifications.

For example,

```
StringOf(383)
```

returns `'  383'`. In this case, `StringOf` used its default formatting settings, a width of 8 characters for integer numbers, since width and precision values weren't specified.

The expression

```
StringOf(3.14:12:4)
```

returns `'  3.1400'` because a width of 12 and a precision of 4 were used.

For a complete description of how `StringOf` handles its parameters, see "Write" on page 120.

To change the default settings `StringOf` uses to format its arguments, see `PLStringOfGet`, `PLStringOfSet`.

Returns `StringOf` returns an item of type `STRING` containing a character representation of its parameters.

See also `PLStringOfGet`, `PLStringOfSet`, `StrOf`, `Write`.

Listing 11.7 StringOf example

```
PROGRAM StringOf_Example;

VAR
  s : STRING[100];
  x : REAL;
  i : INTEGER;

BEGIN
  x := 2.12345;
  i := 4;
  s := StringOf(x:5:5, ' multiplied by ',
```

```
                i:5, ' equals ', (x * i):5:5);
  WriteLn(s);
END. { StringOf_Example }

(* Output:

2.12345 multiplied by     4 equals 8.49380

*)
```

StrOf

Description Used with PLStringOfGet and PLStringOfSet, describes how the StringOf routine formats character strings.

Compatibility ANS Pascal, MPW Pascal, Turbo Pascal

Prototype

```
USES PLStringOf;

TYPE

Justification = (RIGHT, LEFT, MIDDLE);

StrOf = RECORD
  Bools    : RECORD
               cap    : BOOLEAN;
               jst    : Justification;
               width  : LONGINT;
             END; (* Bools *)

  Chars    : RECORD
               cap    : BOOLEAN;
               jst    : Justification;
               width  : LONGINT;
             END; (* Chars *)

  Nbrs     : RECORD
               sign,
               pad    : BOOLEAN;
```

```
                jst    : Justification;
                width  : LONGINT;
              END; (* Nbrs *)

  Packs    : RECORD
                jst    : Justification;
                width  : LONGINT;
              END; (* Packs *)

  Ptrs     : RECORD
                cap,
                pad    : BOOLEAN;
                jst    : Justification;
                width  : LONGINT;
              END; (* Ptrs *)

  Reals    : RECORD
                sign,
                cap,
                pad    : BOOLEAN;
                jst    : Justification;
                width,
                prec   : LONGINT;
              END; (* Reals *)

  Strs     : RECORD
                jst    : Justification;
                width  : LONGINT;
              END; (* Strs *)

{$if not INTEL}
  Comps    : RECORD
                sign,
                pad    : BOOLEAN;
                jst    : Justification;
                width  : LONGINT;
              END; (* Comps *)
{$endif}
END; (* StrOf *)
```

Remarks The `StrOf` record, when used with `PLStringOfGet` and `PLStringOfSet`, describes how the `StringOf` routine formats character strings.

To customize the default settings that `StringOf` uses to format strings, first call the `PLStringOfGet` routine to get the current formatting settings in a record of type `StrOf`, change the fields in the record, then call `PLStringOfSet` with the modified record. Subsequent calls to `StringOf` will use the new settings when formatting items without width or precision specifications.

Subsequent calls to `StringOf` will use the modified settings.

The subrecords within a record of type `StrOf` initialized by `PLStringOfGet` describe how `StringOf` formats character strings to represent various data types. See Table 11.1 for descriptions of the subrecords in `StrOf`.

Table 11.1 Subrecords in StrOf

This subrecord in `StrOf`...	Describes the formatting of this data type
`Bools`	`BOOLEAN`
`Chars`	`CHAR`
`Nbrs`	Integer values
`Packs`	Packed arrays of `CHAR`
`Ptrs`	Hexadecimal values
`Reals`	Floating point and fixed point values
`Strs`	Character strings
`Comps`	`COMP` (available only when generating Mac OS object code).

Fields within each subrecord describe formatting characteristics. Each subrecord has a subset of the fields listed in Table 11.2.

Table 11.2 Subrecord field descriptions within StrOf

This field...	Describes this formatting feature
`cap`	If true, alphabetic characters are converted to uppercase. If false, alphabetic characters are unchanged.
`jst`	If `jst` is `RIGHT`, the characters are right-justified; if `jst` is `LEFT` the characters are left-justified, if `MIDDLE` the characters are centered. Justification only takes place if the item being formatted has fewer than `width` characters.
`width`	If the item being output is fewer than `width` characters, spaces are added based on the `jst` field.
`sign`	If true, a plus sign (+) is output for positive numbers. A minus sign (-) is always output for negative numbers.
`pad`	If true, the value of `jst` is ignored and leading zero characters (0) are output to fill in width characters. If false, the value of `jst` is used and space characters are used to format the item.
`prec`	When set to -1, `StringOf` formats real numeric values as floating point numbers. Otherwise, `StringOf` formats real values as fixed point numbers.

`StrOf` is not part of the Pascal runtime library. To refer to `StrOf`, you must add the `PLStringOf` interface to your program's `USES` clause and add the `PLStringOf.p` file to your project.

See also `PLStringOfGet`, `PLStringOfSet`, `StringOf`, Listing 11.5

Val

Description Converts a character string to a numeric value.

Compatibility Turbo Pascal

Prototype

```
PROCEDURE Val(
  str       : STRING;
  VAR val   : IntegerType ;
  VAR error   : IntegerType);
```

Remarks The `Val` procedure takes a character string as input, `str`, and converts it to an integer value, which it stores in val. On return, the *error* parameter contains the index of the first character that isn't a valid character. If all characters in *str* are valid, *error* contains 0. `Val` ignores tab and space characters.

See also `StringOf`, `Read`

Index

Symbols

$I 35
+ operator 176
+inf 23, 29
<nopage>RTFM
    See happy technical writer 201

A

Abs 20
absolute value 20
adding character strings 176
AND bit 40
ANSI C Standard Library
    assert.h 34
    character strings 36
    errno.h 34
    fenv.h 34
    input/output 35
    locale.h 34
    printf() 35
    printf() variants 35
    scanf() 35
    scanf() variants 35
    setjmp.h 34
    signal.h 34
    stdarg.h 34
    stdio.h 34
    stdlib.h 34
    using call-back routines 36
ANSI C Standard Library. See Metrowerks Standard Library
ANSI.TargetIntel.prefix 35
ANSI.TargetMac.prefix 35
ansi_parms 34
ansi_parms.h 34
Append 76
ARRAY OF CHAR. See PACKED ARRAY OF CHAR.
array operations
    converting from PACKED 61
    converting to PACKED 63
    filling 54
    searching 61
assert 34
assert.h 34
Assign 76
autocloseonquit 137
autoquitonclose 133

B

BAND 38
BAnd 40
BClr 38, 41
bit operations
    AND 40
    check bit 47
    clear 41
    exclusive-OR 49
    high-order word 49, 50
    low-order word 51
    NOT 43
    OR 43
    rotate left 44
    rotate right 45
    set 46
    shift left 46
    shift right 47
    swapping 51
BitAnd 39
BitClr 39
BitNot 39
BitOr 39
BitSet 39
BitShift 39
BitTst 39
BitXor 39
BlockRead 77
BlockWrite 81
BNOT 38, 43
bold 135
Bools 190
BOR 38, 43
BRotL 38, 44
BRotR 38, 45
BSet 38, 46
BSL 38, 46
BSR 38, 47
BTst 38, 47
BXOR 38, 49

byte operations
input/output 83, 85
moving 57, 58, 60
searching 62
ByteRead 83
ByteWrite 85

C

call-back routines. See routines, callback.
character
conversion 163
converting to uppercase 173
end-of-line 72
space 72
tab 133
Chars 190
check bit 47
Chr 163
class membership 149
clear bit 41
Close 86
CodeWarrior IDE
documentation 16
using 16
CodeWarrior Porting Guide 17
CodeWarrior User's Guide 16
column 133
columns 136
Command-Period 130
Comps 190
Concat 176
condense 136
conversion operations
character to number 163
floating point to integer 24, 27, 31
fractional part of real value 23
integer to pointer 171
lower to uppercase 173
pointer to LONGINT 170
real type to LONGINT 27
rounding 27
scalar to LONGINT 170
truncating 24, 31
Copy 177
copy 129
Cos 21
cosine 21
courier 135
ctype 34
ctype.h 34
cut 129

D

deallocating memory 144, 145
Dec 164
decrementing 164
Delete 179
Development environments, other 17
Dispose 144
div_t 34
div_t.h 34

E

Edit menu 129, 130
Edit menu, SIOUX 133
end-of-file 88
end-of-line 90, 111, 127
EOF 88
EOLn 90
Erase 92
errno 34
errno.h 34
even 167
Exclusive-OR bits 49
execution, stopping 130
Exit 65, 66
Exp 22
exponent, natural 22
extend 136

F

fcntl 34
fcntl.h 34
fenv.h 34
File menu 129, 130
File menu, SIOUX 133
FILE. See Type.
FileMode 92

FilePos 93
FileSize 94
FillChar 54
filling an array 54
flip bit 43
float 34
float.h 34
Flush 94
fontface 134, 135
fontid 133, 135
fontsize 134, 135
form feed 100
fp.p 12
Frac 23
fractional part of real value 23
free memory 148, 149
FreeMem 145

G

geneva 135
Get 95
GetMem 146
GetNextEvent 139
GotoXY 129

H

Halt 67
happy technical writer 201
HeapResult 146
helvetica 135
Hi 49
high-order word 49, 50
HiWord 39
HiWrd 39, 50

I

I/O operations
    block input 77
    block output 81
    BOOLEAN output 125
    byte input 83
    byte output 85
    CHAR input 105
    CHAR output 122
    character array output 125
    character string input 105
    checking end-of-file 88
    checking end-of-line 90
    checking status 97
    closing a file 86
    create new file 115
    floating point input 105
    floating point output 124
    integer input 105
    integer output 123
    open existing file 112
    opening for read/write access 99
    page writing 100
    random access 117
    read until end-of-line 111
    reading 104
    record input 95
    record output 104
    scalar input 105
    searching 117
    seeking 117
    STRING output 125
    write with end-of-line 127
    writing 121
Inc 56, 166
incrementing 56, 166
inf 23, 29
initializeTB 133, 139
InOutRes 96
input, standard file 72, 131
input/output 35
Insert 180
Inside Macintosh
    Operating System Utilities 40
Int 24
IOResult 97
IOResult values 98
italic 135

L

largest free block 148
leftpixel 133, 137
Length 182

libraries
console I/O 131
for Mac OS 13, 14
for predefined routines 11
See also routines, predefined.
limits 34
limits.h 34
Ln 24
Lo 51
locale 34
locale.h 34
logarithm, natural 24
LoWord 39
low-order word 51
LoWrd 39, 51

M

Mac OS
documentation. See Inside Macintosh.
initializing 138
Mark 148
math 34
math.h 34
mathematical operations
absolute value 20
cosine 21
logarithm, natural 24
natural exponent 22
others 19
pi, value of 25
pseudo-random number generation 25
pseudo-random number generation initialization 26
sine 28
square 29
square root 30
truncating 24, 31
MaxAvail 148
MemAvail 149
Member 149
memory management
allocating 146, 151
allowing Dispose 157
checking 156
checking available 148, 149
checking status 146
deallocating 144, 145
error handling 157
expanding 158
initializing 155
size of object, getting 159
Metrowerks Standard Library 99
assert.h 34
ctype.h 34
div_t.h 34
fcntl.h 34
float.h 34
limits.h 34
math.h 34
null.h 34
size_t.h 34
stat.h 34
stddef.h 34
string.h 34
time.h 34
unistd.h 34
unix.h 35
utime.h 35
va_list.h 35
wchar_t.h 35
monaco 135
Move 57
MoveLeft 58
MoveRight 60
moving memory 57, 58, 60
MPW Pascal Library 15
MSL 99
MW_Errno.h 98

N

NAN 24, 28, 30, 108
Nbrs 190
New 151
New Project, IDE command 12, 13, 14
newYork 135
NIL 146, 151
normal 135
not bit 43
null 34
null.h 34

O

object membership 149
Odd 167
Open 99
operating systems
    See also Mac OS.
or bit 43
Ord 168
Ord4 170
ordinal operations
    character 163
    decrementing 164
    incrementing 56, 166
    odd or even 167
    ordinality, checking 168, 170
    pointer, converting to 171
    predecessor, getting 171
    See also conversion operations.
    successor, getting 173
    upper case 173
outline 136
output, standard file 72, 131
overflow 20

P

Pack 63
PACKED ARRAY OF CHAR 53, 54, 122, 175
Packs 190
padding, zeros 38
Page 100
Pascal Language settings panel 35
PasLibIntf 15
paste 129
Pi 25
pi, value of 25
PLCrunch 15, 100
PLFilePos 15, 101
PLFlush 15, 101
PLHeapInit 15, 155
PLPurge 15, 102
PLRename 15, 102
PLSetHeapCheck 15, 156
PLSetHeapType 15, 157
PLSetMErrProc 15, 157
PLSetNonCont 15, 158
PLSetVBuf 15, 102
PLStringOf interface 182, 183, 188
PLStringOf.p 182, 183, 188
PLStringOfGet 182
PLStringOfSet 183
+ operator 176
Pointer 171
Pos 184
POSIX. See Metrowerks Standard Library
Pred 171
predefined routines. See routines, predefined.
Prefix File option, Pascal Language panel 35
printf 35
Ptrs 190
Put 104

Q

qualifier identification 12
Quit command, SIOUX 130, 137

R

Random 25
random access 117
random numbers. See mathematical operations
Randomize 26
RandSeed 26
Read 104, 129
ReadLn 111, 129
ReadString 185
Reals 190
Release 158
Rename 112
Reset 112
returning an error code 68
Rewrite 115
rotate bits left 44
rotate bits right 45
Round 12, 27
routines, callback 36
routines, conflicting 12
routines, predefined
    conflicting with other routines 12

declaring 11
libraries 12
referring to 12
scope 12
using as parameters 12
using as parameters. See also type.
rows 133, 136
RunError 68

S

ScanEq 61
scanf 35
ScanNE 62
searching 61, 62, 117
Seek 117
SeekEOF 118
SeekEOLn 119
set a bit 46
setjmp 34
setjmp.h 34
SetTextBuf 120
setupmenus 133
shadow 136
shift bits left 46
shift bits right 47
showstatusline 133, 138
sign extension 38
signal 34
signal.h 34
Sin 28
sine 28
SIOUX
customizing 132
editing text in window 130
handling Macintosh events 139
libraries 131
quit command 130
quitting 130, 133, 137
setting font 135
setting window location 136
settting window size 136
SIOUXSettings 132
status line 131, 133, 138
stopping execution 130
tabs 133, 135
window title 140
SIOUXHandleOneEvent 139
SIOUXSettings 132
SIOUXSetTitle 140
size_t 34
size_t.h 34
SizeOf 159
Sqr 29
SqRt 30
square 29
square root 30
standalone 133
stat 34
stat.h 34
stdarg 34
stdarg.h 34
stddef 34
stddef.h 34
stdio 34
stdio.h 34, 35
stdlib 34
stdlib.h 34
Str 185
string 34
string.h 34
Strs 190
Succ 173
Swap 51
symbol 135
System qualifier 12

T

tabspaces 133, 135
test bit 47
TEXT. See type.
TextBackground 129
TextColor 129
TextMode 130
time 34
time.h 34
times 135
Toolbox
See Mac OS.

ToolUtils.p 39
toppixel 133, 137
transfer, type. See ordinal operations.
trigonometric operations
    cosine 21
    others 19
    sine 28
Trunc 31
Truncate 120
type
    BOOLEAN 105, 122, 125
    CHAR 105, 122
    conversion. See conversion operations.
    DOUBLE 105
    FILE 35, 73
    fixed point 124
    ignoring 53
    INTEGER 105, 122, 123
    LONGINT 105, 123
    LONGREAL 105
    PACKED ARRAY OF CHAR 106, 125
    PACKED ARRAY OF CHAR. See PACKED ARRAY OF CHAR.
    procedural 36
    REAL 105, 122
    scalar 105
    SINGLE 105
    STRING 36, 105, 122, 125
    TEXT 72, 121

U

Unack 61
underline 136
unistd 34
unistd.h 34
Universal Interfaces
    ToolUtils.p 39
unix 35
UNIX. See Metrowerks Standard Library
unix.h 35
UpCase 173
utime 35
utime.h 35

V

va_list 35
va_list.h 35
Val 192

W

WaitNextEvent 139
wchar_t 35
wchar_t.h 35
Win32 API 14
Windows.p 14
Write 121, 129
WriteLn 127, 129

X

XOR 49

Z

zero padding 38

Index

CodeWarrior

Pascal Library Reference

Credits

writing lead: Marc Paquette

other writers: John Roseborough

engineering: Marcel Achim, Nadine Bisson, Michael Cordery, Francois Plouffe

frontline warriors: Bill Catambay, Stephen Chong, James Derrick, Ron Levreault, Ron Liechty, Ray Pierce, Jim Trudeau, Jon Twigge, CodeWarrior users everywhere

Guide to CodeWarrior Documentation

CodeWarrior documentation is modular, like the underlying tools. There are manuals for the core tools, languages, libraries, and targets. The exact documentation provided with any CodeWarrior product is tailored to the tools included with the product. Your product will not have every manual listed here. However, you will probably have additional manuals (not listed here) for utilities or other software specific to your product.

Core Documentation	
IDE User Guide	How to use the CodeWarrior IDE
Debugger User Guide	How to use the CodeWarrior debugger
CodeWarrior Core Tutorials	Step-by-step introduction to IDE components
Language/Compiler Documentation	
C Compilers Reference	Information on the C/C++ front-end compiler
Pascal Compilers Reference	Information on the Pascal front-end compiler
Error Reference	Comprehensive list of compiler/linker error messages, with many fixes
Pascal Language Reference	The Metrowerks implementation of ANS Pascal
Assembler Guide	Stand-alone assembler syntax
Command-Line Tools Reference	Command-line options for Mac OS and Be compilers
Plugin API Manual	The CodeWarrior plugin compiler/linker API
Library Documentation	
MSL C Reference	Function reference for the Metrowerks ANSI standard C library
MSL C++ Reference	Function reference for the Metrowerks ANSI standard C++ library
Pascal Library Reference	Function reference for the Metrowerks ANS Pascal library
MFC Reference	Reference for the Microsoft Foundation Classes for Win32
Win32 SDK Reference	Microsoft's Reference for the Win32 API
The PowerPlant Book	Introductory guide to the Metrowerks application framework for Mac OS
PowerPlant Advanced Topics	Advanced topics in PowerPlant programming for Mac OS
Targeting Manuals	
Targeting BeOS	How to use CodeWarrior to program for BeOS
Targeting Java VM	How to use CodeWarrior to program for the Java Virtual Machine
Targeting Mac OS	How to use CodeWarrior to program for Mac OS
Targeting MIPS	How to use CodeWarrior to program for MIPS embedded processors
Targeting NEC V810/830	How to use CodeWarrior to program for NEC V810/830 processors
Targeting Net Yaroze	How to use CodeWarrior to program for Net Yaroze game console
Targeting Nucleus	How to use CodeWarrior to program for the Nucleus RTOS
Targeting OS-9	How to use CodeWarrior to program for the OS-9 RTOS
Targeting Palm OS	How to use CodeWarrior to program for PalmPilot
Targeting PlayStation OS	How to use CodeWarrior to program for the PlayStation game console
Targeting PowerPC Embedded Systems	How to use CodeWarrior to program for PPC embedded processors
Targeting VxWorks	How to use CodeWarrior to program for the VxWorks RTOS
Targeting Win32	How to use CodeWarrior to program for Windows